Acta Informatica (2020) 57:551–590
https://doi.org/10.1007/s00236-020-00371-w

ORIGINAL ARTICLE

Off-the-shelf automated analysis of liveness properties for just paths

Mark Bouwman[1] · **Bas Luttik**[1] · **Tim Willemse**[1]

Received: 15 April 2019 / Accepted: 3 February 2020 / Published online: 6 May 2020

Abstract
We enrich the operational semantics of a simple process calculus with ACP-style communication with a concurrency relation, so that for every process expression there exists an associated notion of *just path*. We then present sufficient conditions on the communication function and the syntax of process expressions that facilitate the formulation of justness on the level of labels rather than on individual transitions, taking a designated set of signals into account. This paves the way for the formulation of liveness properties under justness assumptions in the modal μ-calculus and their verification on process specifications with the mCRL2 toolset.

1 Introduction

In recent work, van Glabbeek and coauthors suggest that the liveness property for Peterson's mutual exclusion algorithm [17], stating that any process that wants to enter the critical section will eventually enter it, cannot be analysed in CCS and related formalisms [4,7]. This article is the result of our attempt to understand the formal underpinning of this suggestion and its ramifications. In particular, we address the question whether it also implies that the liveness property for Peterson's algorithm cannot be convincingly established by means of a verification with the mCRL2 toolset [2], which has a process-algebra based specification formalism. Before we discuss our contributions, we briefly recap the arguments presented in [4].

1.1 Recap of the arguments in [4]

The authors of [4] note that every process-algebraic specification of a distributed algorithm or system includes unrealistic finite or infinite computations in which some component never makes progress. Since such unrealistic computations typically violate liveness properties, their mere existence is in the way of a proof that all realistic computations do satisfy these properties. Unrealistic computations are then often excluded from consideration by imposing

✉ Bas Luttik
s.p.luttik@tue.nl

[1] Technische Universiteit Eindhoven, Eindhoven, Netherlands

additional assumptions such as *progress* and *fairness* (see [8] for a comprehensive overview of such assumptions).

For the analysis of implementations of so-called *fair schedulers*—of which Peterson's algorithm is an example—one should, however, take care that the fairness assumptions are not too strong, since fair schedulers are, themselves, intended to realise fairness in a system. Van Glabbeek and Höfner [7] have proposed *justness* as a criterion that is just strong enough to exclude unrealistic computation of fair schedulers, but not too strong:

> Once a transition is enabled that stems from a set of parallel components, one (or more) of these components eventually partake in a transition. [8]

It turns out, however, that the proposed notion of justness, when formalised in the context of CCS, still does not exclude certain unrealistic (or at least: unintended) computations of Peterson's algorithm, and some of these computations have liveness violations. The culprit is that in a process-algebraic specification shared variables are components (processes) themselves, and hence reading the value of a shared variable is modelled as an interaction of the component that reads and the component that models the variable. Hence, an infinite computation in which one component continuously wants to assign a new value to the variable, but never actually does, can, nevertheless, be just because another component time and again reads the value of the variable. Yet, in the context of Peterson's algorithm, reading the value of a variable should not be considered to really affect the component corresponding to that variable.

To counteract this problem, it is proposed in [4] to extend the syntax and semantics of CCS with a so-called *signal emission operator*, providing an alternative mechanism to communicate information about the state of a component (e.g., a variable) to other components. Although adding this operator does not increase the absolute expressiveness of the calculus, it does facilitate a refined definition of justness. In this refined definition, the reading of a signal is given special treatment by which computations such as the one described above are not considered just, and thus excluded from consideration. Assuming the refined definition of justness, it is proved in [4] that the specification of Peterson's algorithm in CCS extended with the signal emission operator satisfies the liveness property.

1.2 Our contributions

The signal emission operator is a non-standard process-algebraic construction. It is not part of the specification formalism of mCRL2, nor, to the best of our knowledge, of the specification formalism of any other process-algebra based automated verification tool. The question arises whether the addition of such an operator is essential. If so, a non-trivial overhaul of established verification tools is called for. Our first contribution is to show that it is not, if one is willing to pay a small price: there is no general formal definition of justness for the entire calculus; the formal definition must be tuned to the process expression under consideration. When aiming for an automated verification, this is indeed a negligible price, since one is just interested in the process expression that models the system under verification.

Semantically, the signal emission operator simply adds a self-loop labelled with a *signal* to the state representing the process expression to which it is applied. A signal is just a special type of label, so the self-loop can easily be specified by other means (e.g., using recursion) if a particular subset of the set of labels is designated as signals. Because the choice of an appropriate set of signals depends on how those labels are used in the process expression at hand, the formal definition of justness needs to be specific for a particular process expression.

In the absence of tools supporting the verification under justness of specifications such as Peterson's algorithm, establishing that a specification meets a property remains a manual activity. This is problematic, as the complexity of a typical specification easily leads to cases being missed in the analysis. Therefore, to conduct a convincing automated verification of a property of an algorithm, we not only need to specify the algorithm in a process-algebra based formalism; we also need to formulate the property in a suitable modal logic. Moreover, in the verification of the property, justness has to be taken into account. It is unclear, however, whether this can be achieved without changing the verification algorithms that are used to evaluate the validity of a modal-logic formula with respect to the labelled transition system associated with the process expression. A complication is, for instance, that the definition of justness refers to a notion of *component*, which naturally exists at the level of the syntactic representation of the system (i.e., the process expression), but not at the labelled transition-system level.

Our second contribution is derived from the observation that with the ACP-style communication mechanism [1] of mCRL2, which is more general than the communication mechanism of CCS, Peterson's algorithm can be specified in such a way that justness can be defined referring to labels rather than to components. The idea is to achieve a partitioning of the set of labels that reflects the component structure of the process expression. It is then possible to reformulate justness referring to labels, rather than to components. We generalise the observation regarding Peterson's algorithm and formulate general syntactic conditions that ensure that such a partitioning is possible.

Our third contribution is a template modal μ-calculus formula that expresses a typical liveness property, asserting that on all just paths, an action, say a, is eventually followed by another, say b. This template formula can easily be instantiated by a user wishing to carry out a liveness verification of an algorithm, and only requires information concerning which actions are designated as signals. As a result, standard, off-the-shelf tooling such as mCRL2 can be used to automatically verify liveness properties of algorithms such as Peterson's. In case such verifications fail, evidence [3,18] can be provided, helping the user to pinpoint the root cause.

This paper is organised as follows. In Sect. 2, following [9], we take the notion of labelled transition system with concurrency (LTSC) as technical starting point, and present a definition of justness for it. In Sect. 3 we present a process calculus that is very similar to CCS, except that it has the more general ACP-style communication mechanism. Inspired by the LTSC-semantics that van Glabbeek gives for CCS and its extension with signals in [9], we propose an LTSC-semantics for the process calculus. Then, in Sect. 4 we recapitulate in more detail the argument presented in [4] that Peterson's algorithm cannot be rendered in the process calculus in such a way that all unrealistic paths are excluded by assuming justness. In Sect. 5 we then include a semantic treatment of special labels that take the role of signals. In Sect. 6, we define when an LTSC admits a label-based treatment of justness, proposing a subclass of LTSCs that have a *concurrency-consistent labelling*. In Sect. 7, we present sufficient conditions on process expressions ensuring that the associated LTSC has a concurrency-consistent labelling. Process expressions satisfying these syntactic conditions are amenable to verifications that take justness into account. In Sect. 8 we formalise a general liveness property under justness assumptions for an LTSC that has a concurrency-consistent labelling. In Sect. 9 we comment on the actual verification of the liveness property for Peterson's algorithm with the mCRL2 toolset. In Sect. 10 we present some conclusions.

2 Justness

We recap the definition of labelled transition system with concurrency and the associated notion of just path from [9].

We presuppose disjoint sets $\mathcal{A}$ and $\mathcal{S}$ of *actions* and *signals*, respectively, and let $\mathcal{L} = \mathcal{A} \cup \mathcal{S}$; elements of $\mathcal{L}$ are generally referred to as *labels*. A *labelled transition system* (LTS) is a tuple $(St, Tr, src, target, \ell)$ with St and Tr sets of *states* and *transitions*, respectively, $src, target : Tr \to St$ and $\ell : Tr \to \mathcal{L}$.

We call a transition $t \in Tr$ a *signal transition* if its label is a signal and it does not change state, i.e., if $\ell(t) \in \mathcal{S}$ and $src(t) = target(t)$; otherwise, t is called an *action transition*.

Remark 1 Van Glabbeek mentions in [9] that signal transitions are not supposed to change state, but does not include it as an explicit requirement. Rather, in his work, it is a consequence of the operational semantics of the process calculi under consideration that transitions labelled with signals indeed never change state. The syntax and operational semantics of our process calculus will, by design, admit process specifications that give rise to transitions labelled with signals that do change state. We prefer that such transitions are not treated as signal transitions in the notion of justness. To this end it is convenient to include the requirement explicitly.

Signal transitions are disregarded in the definition of the notion of path. A *path* in a transition system $(St, Tr, src, target, \ell)$ is a finite or infinite alternating sequence $s_0 t_1 s_1 t_2 s_2 \cdots$ of states and action transitions, starting with a state and if it is finite also ending with a state, satisfying $src(t_i) = s_{i-1}$ and $target(t_i) = s_i$ for all relevant i. We say that a state s' is *reachable* from a state s if there exists a path that starts with s and ends with s'. We say that a transition t is *reachable* from a state s if there exists a state s' that is reachable from s and $src(t) = s'$.

Labelled transition systems abstract entirely from the notion of component. For the definition of justness, the notion of component is relevant, at least to the extent that it should be possible to determine that, whenever some transition is enabled, eventually the component (or set of components) from which the transition stems, makes progress. For the formalisation of justness, it turns out to be sufficient to consider labelled transition systems enriched with a concurrency relation on transitions [9]. We first give the formal definition of labelled transition system with concurrency; the requirements on the concurrency relation are explained after the definition.

Definition 2 A *labelled transition system with concurrency* (LTSC) is a tuple $(St, Tr, src, target, \ell, \frown\!\!\bullet)$ consisting of an LTS $(St, Tr, src, target, \ell)$ and a *concurrency relation* $\frown\!\!\bullet \subseteq Tr \times Tr$ such that

1. $\frown\!\!\bullet$ is irreflexive on action transitions (i.e., if t is an action transition, then $t \not\frown\!\!\bullet\, t$), and
2. if t is an action transition and π is a path from $src(t)$ to $s \in St$ such that $t \frown\!\!\bullet\, v$ for all transitions v occurring on π, then there is an action transition u such that $src(u) = s$, $\ell(u) = \ell(t)$ and $t \not\frown\!\!\bullet\, u$.

Intuitively, transitions are *concurrent* if they stem from different (sets of) components, and they *interfere* if they have a component in common. It is then natural to require that the concurrency relation on transitions is irreflexive: a transition cannot be concurrent with itself. Furthermore, if some component (or set of components) can perform some activity, represented by a transition t in the labelled transition system, then after executing transitions concurrent with t—which, by assumption, then stem from different components than t—it

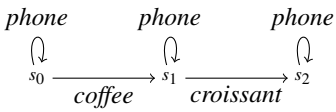
should still be possible for the component to perform that same activity. The activity can be represented by a different transition u in the labelled transition system, but this transition should not be concurrent with t (it should interfere with t, i.e., $t \not\frown u$) and should have the same label.

As explained in [9], justness is a *completeness criterion*: it is used to specify which paths should be considered representing a complete computation of the system. For completeness one wants to distinguish between so-called *blocking* actions and *non-blocking* actions. Intuitively, a blocking action is not entirely under the control of the system that is being specified; it may depend on interaction with the environment. A non-blocking action is thought to be completely under control of the system. A complete computation may end in a state in which only blocking actions are enabled, but not in a state in which non-blocking actions are enabled. The definition of justness takes a set of blocking actions as parameter.

Definition 3 Let $\mathcal{B} \subseteq \mathcal{A}$ be a set of *blocking actions*. A path π in an LTSC is $\mathcal{B}$*-just* if for every action transition t with $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$, a transition u occurs in the suffix of π starting at $src(t)$ such that $t \not\frown u$.

The example below illustrates the concept of justness.

Example 4 Consider a situation in which Alice drinks coffee and eats a croissant in a small cafe, and Bob is engaged in a series of phone calls. The situation can be modelled by the following LTSC:



Suppose that all labels in the above LTSC are non-blocking actions. In case all actions only interfere with themselves, the infinite path consisting of only *phone* transitions from state s_0 is not $\emptyset$-just since the *coffee* transition is enabled in s_0 but no interfering transition is ever taken on this path. In case the *phone* transitions in s_0, s_1 and s_2 *do* interfere with the *coffee* transition and the *croissant* transition—for instance because Bob is also the waiter who serves Alice, preferring to make phone calls instead of taking her orders—then the same infinite path *is* $\emptyset$-just.

3 Process calculus

In [4], the authors claim that information exchanged through signals is essential for the characterisation of just paths in the context of Peterson's algorithm; without signals, paths representing unrealistic executions of Peterson's algorithm are considered just. In [4], justifications for the claim are presented in the context of CCS. First, a version of CCS without signals is considered, Peterson's algorithm is modelled, and then it is shown that justness does not exclude all unrealistic computations. Then, Peterson's algorithm is modelled in a variant of CCS with signals, and it is shown that the corresponding notion of justness works well for Peterson's algorithm. We retrace their steps and in this section introduce a very simple process calculus to specify LTSCs that, as we show in the next section, indeed illustrates the phenomenon observed by the authors. In Sect. 5, we shall also introduce signals, but without changing the syntax of the calculus.

A special feature of our calculus, compared to CCS as considered in [4,9], is that it includes an ACP-style communication mechanism [1]: We presuppose a binary *communication function* on the set of labels $\mathcal{L}$, i.e., a partial function

$$\gamma : \mathcal{L} \times \mathcal{L} \rightharpoonup \mathcal{L}$$

that is

- *commutative*: $\gamma(\lambda_1, \lambda_2)$ is defined if, and only if, $\gamma(\lambda_2, \lambda_1)$ is defined, and if both are defined, then we have $\gamma(\lambda_1, \lambda_2) = \gamma(\lambda_2, \lambda_1)$; and
- *associative*: $\gamma(\lambda_1, \gamma(\lambda_2, \lambda_3))$ is defined if, and only if, $\gamma(\gamma(\lambda_1, \lambda_2), \lambda_3)$ is defined, and if both are defined then we have $\gamma(\lambda_1, \gamma(\lambda_2, \lambda_3)) = \gamma(\gamma(\lambda_1, \lambda_2), \lambda_3)$.

This communication function defines which actions may communicate, and what is the result of that communication. Thus, communication transitions are not all labelled with the same action, as they are in CCS (in CCS all transitions that are the result of communications are labelled with τ). The advantage is that transitions that involve multiple components can be labelled such that from the label it can be determined which components are involved.

We proceed to introduce the syntax of our process calculus and associate an LTSC with it. The LTSC we get is in line with the LTSC that van Glabbeek associates with CCS in [9], though our way of defining it deviates somewhat from van Glabbeek's in [9], as we shall explain below. For now, we presuppose that the set of signals is empty, i.e., $\mathcal{L} = \mathcal{A}$. (In Sect. 5, we shall consider the general case in which the set of signals $\mathcal{S}$ is not empty and adapt the structural operational semantics accordingly.) For the purpose of recursion, we also presuppose a set $\mathcal{I}$ of *agent identifiers*. The set $\mathcal{P}$ of *process expressions* is generated by the following grammar (with A ranging over $\mathcal{I}$, λ ranging over $\mathcal{L}$, and H ranging over subsets of $\mathcal{L}$):

$$P, Q ::= \mathbf{0} \mid \lambda.P \mid P + Q \mid P \parallel Q \mid \partial_H(P) \mid A \ . \tag{1}$$

The constructs $\mathbf{0}$, $\lambda.$ and $+$ are familiar from basic CCS, respectively denoting inaction, action prefix and non-deterministic choice. The construct $\parallel$ stands for ACP-style parallel composition. It represents the arbitrary interleaving of the behaviours of its components, and additionally allows its components to execute communication steps in accordance with the communication function γ: if the left component of the parallel composition can execute label λ_1 and the right component can execute label λ_2 and $\gamma(\lambda_1, \lambda_2)$ is defined, then the parallel composition can execute $\gamma(\lambda_1, \lambda_2)$. The process calculus includes the encapsulation operator ∂_H (similar to the restriction operator in CCS) by which the execution of certain labels can be blocked, and thus communication between components can be enforced. The behaviour of the agent identifiers is defined through a *recursive specification* E, which is a set of defining equations

$$A \stackrel{\text{def}}{=} P \ ,$$

with P a process expression, including precisely one such equation for every $A \in \mathcal{I}$.

We now proceed to associate an LTSC $(St, Tr, src, target, \ell, \smile)$ with our process calculus. The set of states St of this LTSC is the set of process expressions $\mathcal{P}$, as usual. To define a suitable set Tr of transitions, as in [9], we take the collection of derivations in a formal proof system based on the structural operational semantics of the process calculus. We deviate from [9] in how we define the concurrency relation. In [9], van Glabbeek inductively associates a set of *synchrons* with a derivation, which can be thought of as extracting from the derivation all the required component information necessary to define a concurrency relation. We

Table 1 Structural operational semantics

$$\text{(PREF)}\ \frac{}{\lambda.P \xrightarrow{\lambda,\{\epsilon\}} P} \qquad \text{(REC)}\ \frac{P \xrightarrow{\lambda,\alpha} P' \quad A \stackrel{\text{def}}{=} P}{A \xrightarrow{\lambda,\{\epsilon\}} P'}$$

$$\text{(SUM-L)}\ \frac{P \xrightarrow{\lambda,\alpha} P'}{P + Q \xrightarrow{\lambda,\{\epsilon\}} P'} \qquad \text{(SUM-R)}\ \frac{Q \xrightarrow{\lambda,\alpha} Q'}{P + Q \xrightarrow{\lambda,\{\epsilon\}} Q'}$$

$$\text{(PAR-L)}\ \frac{P \xrightarrow{\lambda,\alpha} P'}{P \parallel Q \xrightarrow{\lambda,\mathrm{L}\rhd\alpha} P' \parallel Q} \qquad \text{(PAR-R)}\ \frac{Q \xrightarrow{\lambda,\alpha} Q'}{P \parallel Q \xrightarrow{\lambda,\mathrm{R}\rhd\alpha} P \parallel Q'}$$

$$\text{(COMM)}\ \frac{P \xrightarrow{\lambda_1,\alpha_1} P' \quad Q \xrightarrow{\lambda_2,\alpha_2} Q' \quad \gamma(\lambda_1,\lambda_2) = \lambda}{P \parallel Q \xrightarrow{\lambda,\mathrm{L}\rhd\alpha_1 \cup \mathrm{R}\rhd\alpha_2} P' \parallel Q'} \qquad \text{(ENC)}\ \frac{P \xrightarrow{\lambda,\alpha} P' \quad \lambda \notin H}{\partial_H(P) \xrightarrow{\lambda,\alpha} \partial_H(P')}$$

prefer to annotate the transition relation defined by the structural operational semantics with component information directly.

First, we associate with a process expression P its *static component architecture*, which is determined by the top-level occurrences of $\parallel$ and ∂_H in P. Let $\mathcal{C} = \{\mathrm{L}, \mathrm{R}\}$; we shall refer to a component in a process expression P as a sequence in $\mathcal{C}^*$ (the empty sequence will be denoted by ϵ). We recursively associate with every process expression P a set of *components* $\mathcal{C}(P) \subseteq \mathcal{C}^*$ as follows:

- if $P = \mathbf{0}$, $P = \lambda.P'$ (for some $\lambda \in \mathcal{L}$), $P = P_1 + P_2$, or $P = A$ (for some $A \in \mathcal{I}$), then $\mathcal{C}(P) = \{\epsilon\}$;
- $\mathcal{C}(P_1 \parallel P_2) = \mathrm{L} \rhd \mathcal{C}(P_1) \cup \mathrm{R} \rhd \mathcal{C}(P_2)$, and $\mathcal{C}(\partial_H(P)) = \mathcal{C}(P)$.

(If $X \subseteq \mathcal{C}^*$, then $\mathrm{L} \rhd X = \{\mathrm{L}\sigma \mid \sigma \in X\}$ and $\mathrm{R} \rhd X = \{\mathrm{R}\sigma \mid \sigma \in X\}$.) Note that every $\sigma \in \mathcal{C}(P)$ uniquely identifies a component of P: we denote this component by $P|_\sigma$.

We keep track of which components contribute to a transition in the structural operational semantics for our process calculus, presented in Table 1. It defines a transition relation $\xrightarrow{\lambda,\alpha}$ on process expressions, which is not only endowed with a label $\lambda \in \mathcal{L}$, but also with a set $\alpha \subseteq \mathcal{C}^*$ of components.

The rule (PREF) expresses that a prefix $\lambda.P$ can do a λ-labelled transition to P; furthermore, $\lambda.P$ is by itself a component. So the set of components associated with the transition is $\{\epsilon\}$. The rules (SUM-L) and (SUM-R) express that a non-deterministic choice $P + Q$ can execute a λ-labelled transition from P or from Q. Also $P + Q$ is by itself a component, denoted by ϵ. So the set of components associated with the transition is $\{\epsilon\}$.

The rules (PAR-L), (PAR-R) and (COMM) express, respectively, that a parallel composition $P \parallel Q$ can execute a transition of the components of P, a transition of the components of Q, or execute a transition in which both components of P and Q are involved. In the latter case, the communication function γ must be defined on the labels of the transitions of P and Q and the combined transition is labelled with the result of applying the communication function to these labels. In the case of an application of (PAR-L) or (PAR-R), the sets of components involved in the resulting transitions need to be updated by prefixing all components suitably with L or R, respectively. In the case of an application of (COMM), the involved components of P are prefixed with L, and the involved components of Q are prefixed with R. Finally, the rule (ENC) expresses that ∂_H blocks transitions labelled with $\lambda \in H$; the set of components is simply inherited.

The example below illustrates the operational rules, and how they can be used to construct derivations.

Example 5 The recursive specification given below models the second situation of Example 4, i.e., the situation in which Alice orders coffee and a croissant, and Bob is her waiter.

$$Bob \stackrel{\text{def}}{=} coffee_r.Bob + croissant_r.Bob + phone.Bob \ , \text{ and}$$
$$Alice \stackrel{\text{def}}{=} coffee_s.croissant_s.\mathbf{0} \ ,$$

Assume that γ is a communication function satisfying

$$\gamma(coffee_r, coffee_s) = coffee \text{ and } \gamma(croissant_r, croissant_s) = croissant \ .$$

Then we can derive the following transition with conclusion $Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob$, with source process Bob, target process Bob, and label $coffee_r$:

$$\dfrac{\dfrac{\dfrac{}{coffee_r.Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob}\,(\text{PREF})}{coffee_r.Bob + croissant_r.Bob + phone.Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob}\,(\text{SUM-L})}{Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob}\,(\text{REC})$$

In a similar vein, we can derive a transition that has as conclusion $Alice \xrightarrow{coffee_s,\{\epsilon\}} croissant_s.\mathbf{0}$, and which allows us to derive a transition witnessing the communication that can take place between Alice and Bob:

$$\dfrac{\dfrac{\vdots}{Bob \xrightarrow{coffee_r,\{\epsilon\}} Bob} \qquad \dfrac{\vdots}{Alice \xrightarrow{coffee_s,\{\epsilon\}} croissant_s.\mathbf{0}}}{Bob \parallel Alice \xrightarrow{coffee,\{\text{L,R}\}} Bob \parallel croissant_s.\mathbf{0}}\,(\text{COMM})$$

The above derivation shows that both Alice and Bob contribute equally to the transition that results in Alice drinking a cup of coffee.

Now we let Tr be the set of all derivations[1] that can be constructed using the structural operational rules in Table 1, and we define src, $target$ and ℓ by stipulating that if $t \in Tr$ is a derivation and $P \xrightarrow{\lambda,\alpha} P'$ is its conclusion, then $src(t) = P$, $target(t) = P'$ and $\ell(t) = \lambda$. Furthermore, we write $comp(t)$ to denote the set of components α contributing to t.

It remains to define the concurrency relation $\frown$. We define that transitions t and u are concurrent (notation: $t \frown u$) if $comp(t) \cap comp(u) = \emptyset$, i.e., if none of the components contributing to t are contributing to u.

Lemma 6 *For all transitions t and v, if $src(t) = src(v)$ and $t \frown v$, then there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $comp(u) = comp(t)$.*

Proof By induction on v; see Lemma 44 in "Appendix A" for details. □

Proposition 7 *The structure $\mathbf{P} = (St, Tr, src, target, \ell, \frown)$ with components as defined above is an LTSC.*

[1] The notion of derivation with respect to a set of derivation rules can be defined inductively as usual; we omit it here.

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 $readyB := false$

Fig. 1 Peterson's algorithm (pseudocode)

Proof From the rules in Table 1 it is immediate that whenever $P \xrightarrow{\lambda,\alpha} P'$, then $\alpha \neq \emptyset$. So for every $t \in Tr$ we have that $comp(t) \cap comp(t) = \alpha \neq \emptyset$. It follows that $t \not\smile t$ and hence $\smile$ is irreflexive. That $\smile$ also satisfies the second requirement of Definition 2 follows with a straightforward induction on the length of π using Lemma 6. □

4 Modelling Peterson's algorithm

Peterson's algorithm for mutual exclusion provides a classical solution to enable two processes to use a shared resource in a mutually exclusive manner. In the algorithm, the shared resource is referred to as the *critical section*. The algorithm ensures that at all times only one of the two processes is in the critical section. A desired liveness property of a mutual exclusion algorithm is that whenever one of the two processes wishes to enter the critical section, then it will eventually do so. In this section, we shall discuss how Peterson's algorithm can be modelled in the process calculus introduced in the previous section. Then, we shall recap the argument, already presented in [4], that the notion of justness associated with the process calculus is too weak to exclude all unrealistic paths violating the liveness property. In the next section, we shall refine the definition of justness in order to facilitate an exhaustive verification under this notion of justness of the aforementioned liveness property using the mCRL2 toolset.

Peterson's algorithm is shown in Fig. 1. Processes A and B communicate via shared variables. By setting Boolean variables *readyA* and *readyB*, respectively, they signal to the other process their wish to enter the critical section. In addition, a shared variable *turn* is used to keep track of whose turn it is to enter the critical section next; the idea is that a process, before entering its critical section, courteously always first grants access to the other process. This way of using *turn* is essential for ensuring both deadlock freedom and mutual exclusion.

In a message-passing process calculus, global variables are modelled as separate processes with which other processes can interact. Processes modelling a variable keep track of the value of the variable and can communicate with other processes in read and write operations. In our model, to read a variable, the variable that is being read performs an action $s\_rd_{var}^{val}$ and the process that reads the variable performs an action $r\_rd_{var}^{val}$. Together they communicate to a transition labelled with rd_{var}^{val}. A similar communication, labelled with $asgn_{var}^{val}$, is defined to write to a variable. To cover all the interactions with variables in Peterson's algorithm we define the communication function γ in such a way that it satisfies the following equations and is undefined otherwise:

$$\begin{array}{ll}
\gamma(r_asgn_P^b, s_asgn_P^b) = asgn_P^b & (b \in \{true, false\},\ P \in \{RA, RB\}),\\
\gamma(r_rd_P^b, s_rd_P^b) = rd_P^b & (b \in \{true, false\},\ P \in \{RA, RB\}),\\
\gamma(r_asgn_T^t, s_asgn_T^t) = asgn_T^t & (t \in \{A, B\}), \text{ and}\\
\gamma(r_rd_T^t, s_rd_T^t) = rd_T^t & (t \in \{A, B\}).
\end{array} \tag{2}$$

We model the behaviour of the three variables *readyA*, *readyB* and *turn* with process identifiers RA^b, RB^b and T^t (with the superscripts referring to the current value of the variable), defined by the following equations:

$$\begin{array}{l}
RA^b = r_asgn_{RA}^{true}.RA^{true} + r_asgn_{RA}^{false}.RA^{false} + s_rd_{RA}^b.RA^b \quad (b \in \{true, false\}),\\
RB^b = r_asgn_{RB}^{true}.RB^{true} + r_asgn_{RB}^{false}.RB^{false} + s_rd_{RB}^b.RB^b \quad (b \in \{true, false\}), \text{ and}\\
T^t = r_asgn_T^A.T^A + r_asgn_T^B.T^B + s_rd_T^t.T^t \quad (t \in \{A, B\}).
\end{array}$$

Our specification uses labels **noncritA**, **noncritB**, **critA**, **critB**, to represent exiting the noncritical and critical sections, respectively. Process identifiers *procA* and *procB* model the behaviour of processes A and B. They are defined by the following equations [using the abbreviation $(\lambda_1 + \lambda_2).P$ for $\lambda_1.P + \lambda_2.P$]:

$$\begin{array}{l}
procA = \mathbf{noncritA}.s_asgn_{RA}^{true}.s_asgn_T^B.(r_rd_{RB}^{false} + r_rd_T^A).\mathbf{critA}.s_asgn_{RA}^{false}.procA, \text{ and}\\
procB = \mathbf{noncritB}.s_asgn_{RB}^{true}.s_asgn_T^A.(r_rd_{RA}^{false} + r_rd_T^B).\mathbf{critB}.s_asgn_{RB}^{false}.procB.
\end{array}$$

Together, the process definitions form the recursive specification E_{Pet} consisting of eight process identifiers: *procA*, *procB*, RA^{true}, RA^{false}, RB^{true}, RB^{false}, T^A and T^B. With the set H defined by

$$\begin{array}{rl}
H = \{s_asgn_P^b, r_asgn_P^b, s_rd_P^b, r_rd_P^b \mid b \in \{true, false\},\ P \in \{RA, RB\}\}\\
\qquad \cup \{s_asgn_T^t, r_asgn_T^t, s_rd_T^t, r_rd_T^t \mid t \in \{A, B\}\},
\end{array}$$

we can now specify Peterson's algorithm with the process expression

$$Pet = \partial_H(procA \parallel (procB \parallel (RA^{false} \parallel (RB^{false} \parallel T^A)))) \ .$$

Remark 8 Our specification of Peterson's algorithm is almost identical to the CCS model presented in [4]. The difference is in how communication is defined. CCS presupposes a standard communication function by which an action a can communicate with its co-named action $\bar{a}$, resulting in a special action τ. In our setting, the exact same behaviour as defined by the specification in [4] would be obtained by using, instead of the communication function γ defined above, a communication function γ_{CCS} defined by

$$\begin{array}{ll}
\mathrm{CCS}(r_asgn_P^b, s_asgn_P^b) = \tau & (b \in \{true, false\},\ P \in \{RA, RB\}),\\
\mathrm{CCS}(r_rd_P^b, s_rd_P^b) = \tau & (b \in \{true, false\},\ P \in \{RA, RB\}),\\
\mathrm{CCS}(r_asgn_T^t, s_asgn_T^t) = \tau & (t \in \{A, B\}), \text{ and}\\
\mathrm{CCS}(r_rd_T^t, s_rd_T^t) = \tau & (t \in \{A, B\}).
\end{array} \tag{3}$$

To get an appropriate notion of just path starting from *Pet*, we define the set of blocking actions.

$$\mathcal{B} = \{\mathbf{noncritA}, \mathbf{noncritB}\}$$

Let π denote the unique path starting with *Pet* such that if all states are omitted from it then we obtain the following sequence of labels:

$$\mathbf{noncritA}.(\mathbf{noncritB}.asgn_{RB}^{true}.asgn_T^A.rd_{RA}^{false}.\mathbf{critB}.asgn_{RB}^{false})^\infty \ .$$

The path π violates the liveness criterion as process A wants to enter the critical section but is never able to, waiting to write to the variable *readyA*. It is deemed unrealistic, as process B reading *readyA* intuitively cannot prevent process A from writing it. To assess whether π is just we need to examine whether for every action transition t with $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$, a transition u occurs in the suffix of π starting at $src(t)$ such that $t \not\prec\!\!\bullet\, u$. The only component of interest here is $procA$ as all other components partake in infinitely many transitions. Let t denote some transition labelled with $asgn_{RA}^{true}$, with $src(t) \in \pi$. There always exists a transition u labelled with rd_{RA}^{false} in the suffix of π starting at $src(t)$. The components partaking in t are L and RRL and the components partaking in u are RL and RRL. Hence, due to the overlap, $t \not\prec\!\!\bullet\, u$; the path violating the liveness property is just.

A more refined definition of the concurrency relation is needed to specify that certain interactions, such as reading a variable, do not interfere with other interactions with the same component. This requires distinguishing between components contributing passively to a transition and components really affected by a transition.

5 Signals

In the previous section it was observed that the specification of Peterson's algorithm in the proposed process calculus does not yield the appropriate notion of just path, at least not with the given semantics. The culprit is a combination of two aspects. First, shared variables need to be modelled as separate processes. Second, the process calculus does not offer a facility to distinguish between the activities of reading and writing a variable while, intuitively, if some component reads the value of a variable then this should not prevent another process from writing a new value to it.

The solution proposed in [4] is to extend the syntax of CCS with a *signal emission operator*, in order to treat signals differently in the definition of the concurrency relation. A separate set $\mathcal{S}$ of signals is presupposed, and the signal emission operator adds a λ-labelled self-loop to a state if it can emit signal $\lambda \in \mathcal{S}$. Variables, modelled as processes, then emit their values in the form of signals, and reading the value of a variable can then be treated as not affecting the variable. As a consequence, paths on which some component wants to write to a variable but never succeeds because the variable is perpetually read by some other component is not considered just.

Adding a signal emission operator solves the problem uniformly: with every process expression of the process calculus an appropriate notion of just path is associated: if a component only contributes to a transition by emitting a signal, then this contribution is considered passive. A disadvantage of the solution, however, is that it requires an addition to the syntax of the calculus. As a consequence, standard verification technology such as the mCRL2 toolset, which does not include a signal emission operator, cannot be used to perform verifications taking justness into account.

Here we opt for a different solution, which does not require an addition to the syntax of the process calculus. Instead, it suffices to distinguish a separate set of signals $\mathcal{S}$ and tune the notion of justness to take signals into account. We need to modify the structural operational semantics, giving signals a special status: whenever a transition labelled with a signal indeed does not change state, then it is considered to be a signal. But this modification of the structural operational semantics is only necessary to get an appropriate definition of the concurrency relation. In Sects. 6 and 7, we shall propose sufficient conditions on a process expression (and the underlying recursive specification) that ensure that all transitions labelled

Table 2 Structural operational semantics taking signals into account

$$(\textsc{Pref})\ \frac{}{\lambda.P \xrightarrow{\lambda,\{\epsilon\},\emptyset} P}$$

$$(\textsc{Rec})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P' \quad A \stackrel{\text{def}}{=} P}{A \xrightarrow{\lambda,\alpha',\varsigma'} P'} \begin{cases} \alpha' = \emptyset,\ \varsigma' = \{\epsilon\} \text{ if } \lambda \in \mathcal{S} \text{ and } P' = A \\ \alpha' = \{\epsilon\},\ \varsigma' = \emptyset \text{ otherwise} \end{cases}$$

$$(\textsc{Sum-L})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P'}{P + Q \xrightarrow{\lambda,\alpha',\varsigma'} P'} \begin{cases} \alpha' = \emptyset,\ \varsigma' = \{\epsilon\} \text{ if } \lambda \in \mathcal{S} \text{ and } P' = P + Q \\ \alpha' = \{\epsilon\},\ \varsigma' = \emptyset \text{ otherwise} \end{cases}$$

$$(\textsc{Sum-R})\ \frac{Q \xrightarrow{\lambda,\alpha,\varsigma} Q'}{P + Q \xrightarrow{\lambda,\alpha',\varsigma'} Q'} \begin{cases} \alpha' = \emptyset,\ \varsigma' = \{\epsilon\} \text{ if } \lambda \in \mathcal{S} \text{ and } Q' = P + Q \\ \alpha' = \{\epsilon\},\ \varsigma' = \emptyset \text{ otherwise} \end{cases}$$

$$(\textsc{Par-L})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P'}{P \parallel Q \xrightarrow{\lambda,\mathrm{L}\rhd\alpha,\mathrm{L}\rhd\varsigma} P' \parallel Q} \qquad (\textsc{Par-R})\ \frac{Q \xrightarrow{\lambda,\alpha,\varsigma} Q'}{P \parallel Q \xrightarrow{\lambda,\mathrm{R}\rhd\alpha,\mathrm{R}\rhd\varsigma} P \parallel Q'}$$

$$(\textsc{Comm})\ \frac{P \xrightarrow{\lambda_1,\alpha_1,\varsigma_1} P' \quad Q \xrightarrow{\lambda_2,\alpha_2,\varsigma_2} Q' \quad \gamma(\lambda_1,\lambda_2) = \lambda}{P \parallel Q \xrightarrow{\lambda,\mathrm{L}\rhd\alpha_1\cup\mathrm{R}\rhd\alpha_2,\mathrm{L}\rhd\varsigma_1\cup\mathrm{R}\rhd\varsigma_2} P' \parallel Q'}$$

$$(\textsc{Enc})\ \frac{P \xrightarrow{\lambda,\alpha,\varsigma} P' \quad \lambda \notin H}{\partial_H(P) \xrightarrow{\lambda,\alpha,\varsigma} \partial_H(P')}$$

with signals are indeed signal transitions. This, in combination with the use of an appropriate communication function that preserves component information, will eventually obviate the need for explicitly defining a concurrency relation on transitions, because it can be deduced from the labelling.

Henceforth we allow $\mathcal{S}$ to be non-empty. The syntax of the process calculus [see (1) on p. 6] remains the same. In the structural operational semantics, however, we distinguish between components contributing actively and components contributing passively to a transition. A component contributes passively to a transition if another component reads one of its signals, i.e., the component participates with a transition that is labelled with a signal and this transition does not change the state of the component. The modified structural operational semantics in Table 2 defines a transition relation $\xrightarrow{\lambda,\alpha,\varsigma}$ on process expressions, which is endowed with a label $\lambda \in \mathcal{L}$, a set $\alpha \subseteq \mathcal{C}^*$ of *active components* and a set $\varsigma \subseteq \mathcal{C}^*$ of *signalling components*.

Note that $\lambda.P \neq P$, and therefore a transition emanating from a prefix always changes state. Thus, according to the rule (PREF), the transition from a prefix has an active component ϵ and no signalling components.

If an identifier A is the source of a transition that has A also as its target, and this transition is labelled with a signal, then this transition has a signalling component ϵ and no active components; otherwise, the transition has an active component ϵ and no signalling components.

Due to the presence of recursion, it may also happen that $P + Q$ is both the source and the target of a transition, and if such a transition is labelled with a signal, then we want to treat it as a signal transition. This is reflected in (SUM-L) and (SUM-R) by distinguishing whether the target of the transition equals $P + Q$ and is labelled with a signal: if so, then the transition has no active components and a signalling component ϵ; otherwise, the transition has an active component ϵ and no signalling components.

In an application of (PAR-L), both the active and signalling components of the premise are prefixed with an L; in an application of (PAR-R), they are prefixed with an R; in an application of (COMM) the components of the left premise are prefixed with an L and those of the right premise are prefixed with an R. In an application of (ENC), both the sets of active and signalling components are simply inherited from the premise.

Example 9 Consider the recursive specification of Peterson's algorithm—and in particular the specification of RA^{false}—given in the previous section. Suppose that $s\_rd_{RA}^{false} \in \mathcal{S}$ but $rd_{RA}^{false}, r\_rd_{RA}^{false} \notin \mathcal{S}$. Then we have the following (fragment of a) derivation:

$$\dfrac{\dfrac{\vdots}{RA^{false} \xrightarrow{s\_rd_{RA}^{false},\emptyset,\{\epsilon\}} RA^{false}}(\textsc{Rec}) \qquad \dfrac{}{r\_rd_{RA}^{false}.\mathbf{0} \xrightarrow{r\_rd_{RA}^{false},\{\epsilon\},\emptyset} \mathbf{0}}(\textsc{Pref})}{RA^{false} \parallel r\_rd_{RA}^{false}.\mathbf{0} \xrightarrow{rd_{RA}^{false},\{\mathrm{R}\},\{\mathrm{L}\}} RA^{false} \parallel \mathbf{0}}(\textsc{Comm})$$

The component RA^{false} contributes a signal transition, and hence does not actively contribute to the communication. As a consequence, the path we identified earlier as constituting a liveness violation of Peterson's algorithm is, with the revised semantics, no longer just.

We now associate a revised LTSC with our process calculus as follows. Its set of states St is again the set of process expressions. Its set of transitions Tr is the set of all derivations in accordance with the new structural operational semantics in Table 2. Again, if $t \in Tr$ is a derivation with conclusion $P \xrightarrow{\lambda,\alpha,\varsigma} P'$, then $src(t) = P$, $target(t) = P'$ and $\ell(t) = \lambda$. We define the concurrency relation using a refined notion of component, in which we distinguish between *necessary participants* and *affected* components. The set of necessary participants of a transition t, denoted by $npc(t)$, is defined as

$$npc(t) = \alpha \cup \varsigma \ ,$$

and the set of *affected components* of t, denoted by $afc(t)$, is defined as

$$afc(t) = \alpha \ .$$

We define that transitions t and u are concurrent (notation: $t \frown\!\!\bullet\, u$) if none of the components necessary for t are affected by u, i.e., if $npc(t) \cap afc(u) = \emptyset$.

To satisfy the requirements on $\frown\!\!\bullet$ that it is irreflexive on action transitions, it is important that the set of affected components $afc(t)$ of an action transition t is non-empty, for otherwise $npc(t) \cap afc(t) = \emptyset$. The following example illustrates that we need to formulate some mild restrictions on the communication function for this.

Example 10 Consider the recursive specification consisting of the following two defining equations:

$$A \stackrel{\text{def}}{=} \lambda_1.A \ , \text{ and}$$
$$B \stackrel{\text{def}}{=} \lambda_2.B \ ,$$

and suppose that γ is a communication function satisfying

$$\gamma(\lambda_1, \lambda_2) = \gamma(\lambda_2, \lambda_1) = \lambda_3 \ .$$

Furthermore, suppose that $\lambda_1, \lambda_2 \in \mathcal{S}$, while $\lambda_3 \in \mathcal{A}$. Then we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\lambda_1.A \xrightarrow{\lambda_1,\{\epsilon\},\emptyset} A}(\textsc{Pref})}{A \xrightarrow{\lambda_1,\emptyset,\{\epsilon\}} A}(\textsc{Rec}) \quad \dfrac{\dfrac{}{\lambda_1.B \xrightarrow{\lambda_2,\{\epsilon\},\emptyset} B}(\textsc{Pref})}{B \xrightarrow{\lambda_2,\emptyset,\{\epsilon\}} B}(\textsc{Rec})}{A \parallel B \xrightarrow{\lambda_3,\emptyset,\{\mathrm{L},\mathrm{R}\}} A \parallel B}(\textsc{Comm})$$

Since $\lambda_3 \in \mathcal{A}$, this derivation is an action transition, but the set of affected components is empty. The culprit in this example is that communication between the two signals λ_1 and λ_2 results in an action λ_3.

We can exclude the situation as described in the preceding example by requiring that the communication of two signals never results in an action. It is convenient and natural to also require the converse: the communication of an action with another label should never result in a signal.

Definition 11 A communication function γ is *signal-respecting* if $\gamma(\lambda_1, \lambda_2) \in \mathcal{S}$ if, and only if, $\lambda_1, \lambda_2 \in \mathcal{S}$.

Lemma 12 *If the communication function γ is signal-respecting, then a transition t is a signal transition if, and only if, $afc(t) = \emptyset$.*

Proof By induction on t; see Lemma 45 in "Appendix A" for details. □

In the following corollary, which is an immediate consequence of the preceding lemma, we establish that $\frown\!\!\bullet$ satisfies condition 1 of Definition 2.

Corollary 13 *If the communication function γ is signal-respecting, then $\frown\!\!\bullet$ is irreflexive on action transitions, i.e., for all action transitions t we have $t \not\frown\!\!\bullet\, t$.*

Proof Let t be an action transition. Then, by Lemma 12, $afc(t) \neq \emptyset$. Since $afc(t) \subseteq npc(t)$, it follows that $npc(t) \cap afc(t) \neq \emptyset$, and hence $t \not\frown\!\!\bullet\, t$. □

Lemma 14 *For all transitions t and v, if $src(t) = src(v)$ and $npc(t) \cap afc(v) = \emptyset$, then there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $npc(u) = npc(t)$. If γ is signal-respecting and t is an action transition, then so is u.*

Proof By induction on v; see Lemma 46 in "Appendix A" for details. □

It follows from the preceding lemma that the relation $\frown\!\!\bullet$ associated with our process calculus satisfies condition 2 of Definition 2, as established in the following corollary.

Corollary 15 *If γ is signal-respecting, t is an action transition and π is a path from $src(t)$ to some process expression P such that $t \frown\!\!\bullet\, v$ for all transitions v occurring on π, then there is an action transition u such that $src(u) = P$, $\ell(u) = \ell(t)$ and $t \not\frown\!\!\bullet\, u$.*

Proof Straightforward induction on the length of π using Lemma 14. □

From Corollaries 13 and 15 we get the following proposition.

Proposition 16 *Let γ be signal-respecting, let $St = \mathcal{P}$, let Tr be the set of all derivations of transitions in accordance with the operational semantics, stipulating that if $t \in Tr$ is a derivation with conclusion $P \xrightarrow{\lambda,\alpha,\varsigma} P'$, then $src(t) = P$, $target(t) = P'$ and $\ell(t) = \lambda$, $npc(t) = \alpha \cup \varsigma$ and $afc(t) = \alpha$, and defining $\rightsquigarrow$ by $t \rightsquigarrow u$ if, and only if, $npc(t) \cap afc(u) = \emptyset$. Then*

$$\mathbf{P} = (St, Tr, src, target, \ell, \rightsquigarrow)$$

is an LTSC.

Example 17 Returning to the running example of Peterson's algorithm we reconsider the path that violates liveness. First, we define the signal actions and check whether the communication function is signal-respecting.

$$\mathcal{S} = \{s\_rd_P^b \mid b \in \{true, false\},\ P \in \{RA, RB\}\} \cup \{s\_rd_T^t \mid t \in \{A, B\}\}$$

It is easy to see that the communication function γ, defined in Eq. (2) on p. 9, is signal-respecting. Taking the LTSC as defined in Proposition 16 we re-examine the liveness violating path π presented at the end of Sect. 4, which gives rise to the following sequence of labels:

$$\mathbf{noncritA}.(\mathbf{noncritB}.asgn_{RB}^{true}.asgn_T^A.rd_{RA}^{false}.\mathbf{critB}.asgn_{RB}^{false})^\infty.$$

Let t and u be any two transitions with labels $asgn_{RA}^{true}$ and rd_{RA}^{false}, respectively. Then $npc(t) = \{\text{L}, \text{RRL}\}$ and $afc(u) = \{\text{RL}\}$. Therefore $npc(t) \cap afc(u) = \emptyset$ and thus $t \rightsquigarrow u$. We conclude that path π contains transition t, $src(t) \in \pi$, for which there does not exist a transition v in the suffix of π such that $t \not\rightsquigarrow v$. The path π is therefore not just and can be ruled out. Note that this does not constitute a proof of liveness, we have only reasoned about a single path. To prove liveness we need to prove that there does not exist another liveness violating path that is just.

6 Concurrency-consistent labelling

The semantics we associated with our process calculus in the previous section enables reasoning about just paths without the need for additional operators in the language. This allows one to manually analyse, e.g., the required liveness property of Peterson's algorithm in a standard process algebra, by reasoning directly about the relevant just paths in the LTSC under analysis. Our aim, however, is to facilitate the automated verification of liveness properties for just paths, using toolsets such as mCRL2. Such toolsets are based on labelled transition systems without a concurrency relation. Moreover, in these toolsets, properties need to be expressed in a modal logic that has modalities that refer to labels, and not to individual transitions.

Our specification of Peterson's algorithm is such that it allows a characterisation of its just paths in terms of labels rather than referring to individual transitions in the LTSC. This is possible, because the labelling of transitions reachable from *Pet* is consistent with the concurrency relation on those transitions.

In this section, we formally define when an LTSC has a concurrency-consistent labelling, and we prove that LTSCs with a concurrency-consistent labelling allow a characterisation of just paths in terms of labels instead of individual transitions. In the next section, we shall provide a sufficient syntactic criterion on specifications in our process calculus that ensure that the associated LTSC has a concurrency-consistent labelling, and we argue that our specification of Peterson's algorithm satisfies this syntactic criterion.

Definition 18 An LTSC $(St, Tr, src, target, \ell, \rightsquigarrow)$ has a *concurrency-consistent labelling* if for every $t \in Tr$, $\ell(t) \in \mathcal{S}$ implies $src(t) = target(t)$, and there exists a binary relation $\rightsquigarrow$ on the set of labels $\mathcal{L}$ such that for all transitions $t, u \in Tr$ we have that $t \rightsquigarrow u$ if, and only if, $\ell(t) \rightsquigarrow \ell(u)$.

Clearly, there is no harm in the overloading of the symbol $\rightsquigarrow$. In an LTSC with a concurrency-consistent labelling the relation on $\mathcal{L}$ is uniquely determined by the relation on Tr. Furthermore, it will be clear from the context whether we mean the relation on transitions or the relation on labels. For an LTSC with concurrency-consistent labelling, we can reformulate the notion of $\mathcal{B}$-justness referring to labels instead of transitions. A label $\lambda \in \mathcal{L}$ is *enabled* in a state $s \in St$ if there is a transition t with $src(t) = s$ and $\ell(t) = \lambda$. An action $\lambda \in \mathcal{A}$ is *eliminated* on a path π if there is a transition t on π such that $\lambda \not\rightsquigarrow \ell(t)$. In an LTSC with a concurrency-consistent labelling, action transitions are not labelled by signals, so a non-blocking action transition is labelled by an element of the complement $\overline{\mathcal{B}} = \mathcal{A} \backslash \mathcal{B}$ of $\mathcal{B}$ relative to $\mathcal{A}$.

Proposition 19 *Let $\mathcal{B} \subseteq \mathcal{A}$ be a set of blocking actions. If $(St, Tr, src, target, \ell, \rightsquigarrow)$ has a concurrency-consistent labelling, then a path π is $\mathcal{B}$-just if, and only if, for every state s on π and every $\lambda \in \overline{\mathcal{B}}$ enabled in s, λ is eliminated in the suffix of π starting at s.*

Proof Let π be a path in $(St, Tr, src, target, \ell, \rightsquigarrow)$.

To prove the implication from left to right, suppose that π is $\mathcal{B}$-just and suppose that $\lambda \in \overline{\mathcal{B}}$ is enabled in some state s on π. Then there is an action transition t with $src(t) = s$ and $\ell(t) = \lambda$, so, by $\mathcal{B}$-justness, a transition u occurs in the suffix of π starting at $src(t) = s$ such that $t \not\rightsquigarrow u$. Since the LTSC has a concurrency-consistent labelling, it follows that $\lambda = \ell(t) \not\rightsquigarrow \ell(u)$, and hence λ is eliminated on the suffix of π starting at s.

To prove the implication from right to left, let t be an action transition such that $\ell(t) \notin \mathcal{B}$ and $src(t) \in \pi$. Then $\ell(t)$ is enabled and, since t is an action transition and the LTSC has a concurrency-consistent labelling, it follows that $\ell(t) \in \overline{\mathcal{B}}$, so λ is eliminated in the suffix of π starting at $src(t)$. So there is a transition u in the suffix of π starting at $src(t)$ such that $\ell(t) \not\rightsquigarrow \ell(u)$. Hence, since the LTSC has a concurrency-consistent labelling, $t \not\rightsquigarrow u$, confirming that π is $\mathcal{B}$-just. □

7 Specifying an LTSC with concurrency-consistent labelling

The LTSC $\mathbf{P}$ associated with the process calculus in Sect. 5 does not have a concurrency-consistent labelling, simply because there exist process expressions (e.g., $\lambda.\mathbf{0}$ with $\lambda \in \mathcal{S}$) that give rise to state-changing transitions labelled with signals. In automated verification, however, we are often only interested in the restriction of $\mathbf{P}$ to the set of process expressions reachable from some initial process expression; for example, when verifying Peterson's algorithm we are only interested in states and transitions reachable from *Pet*. We shall now first formally define the LTSC associated with a process expression P, and then formulate sufficient syntactic conditions that guarantee that this LTSC has a concurrency-consistent labelling.

Definition 20 Let P be a process expression. The *LTSC associated with P* has as set of states the set of all process expressions reachable from P in $\mathbf{P}$, as transitions the set of all transitions reachable from P, and functions *src*, *target*, ℓ and relation $\rightsquigarrow$ obtained by restricting those of $\mathbf{P}$ to the set of transitions reachable from P.

In Sect. 5, the concurrency relation $\smile\!\!\bullet$ on transitions was derived from assignments $npc : Tr \to 2^{\mathcal{C}^*}$ and $afc : Tr \to 2^{\mathcal{C}^*}$ of necessary participants and affected components to individual transitions. It is convenient to formulate sufficient conditions in terms of assignments $npc_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ and $afc_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ of necessary and affected components to labels, respectively, satisfying for every transition t

$$npc(t) = npc_\ell(\ell(t)), \text{ and} \tag{4}$$
$$afc(t) = afc_\ell(\ell(t)) \ . \tag{5}$$

It is not possible to satisfy these equations in general: an appropriate assignment of components to labels largely depends on the process expression under consideration. Moreover, it may not even be possible to define $npc_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ and $afc_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ in such a way that the equations above are satisfied for all reachable transitions.

Example 21 Consider the specification *Pet* of Peterson's algorithm presented in Sect. 4, and consider the state reached from *Pet* by first executing **noncritA** and then executing **noncritB**. In that state, two transitions are enabled: let us denote by t the transition corresponding to the activity of process A assigning the value *true* to the variable *readyA* (this is statement ℓ_2 in Fig. 1) and let us denote by u the transition corresponding to the activity of process B assigning the value *true* to the variable *readyB* (this is statement m_2 in Fig. 1). Then $npc(t) = \{\text{L}, \text{RRL}\}$ and $npc(u) = \{\text{RL}, \text{RRRL}\}$. Now observe that, in the context of the CCS communication function γ_{CCS}, defined in Eq. (3) on p. 10, we have that $\ell(t) = \ell(u) = \tau$, and hence it is not possible to define a mapping $npc_\ell : \mathcal{L} \to 2^{\mathcal{C}^*}$ satisfying (4). Note that with the communication function γ, defined in Eq. (2) on p. 9 the problem disappears, since t and u have distinct labels $asgn^{true}_{RA}$ and $asgn^{true}_{RB}$, respectively.

The goal in this section is to formulate sufficient conditions on the communication function γ and a process expression P that allow us to define npc_ℓ and afc_ℓ satisfying (4) and (5) for all transitions t reachable from P. Furthermore, we show that our specification of Peterson's algorithm satisfies these restrictions.

We first formulate some basic requirements on npc_ℓ and afc_ℓ, expressing that the set of affected components associated with a label is included in the set of necessary components, and that signals do not have active components.

Definition 22 Let $C \subseteq \mathcal{C}^*$ be a finite set of static components. A C-assignment is a pair (npc_ℓ, afc_ℓ) of mappings $npc_\ell, afc_\ell : \mathcal{L} \to 2^C$ such that

1. $afc_\ell(\lambda) \subseteq npc_\ell(\lambda)$ for all $\lambda \in \mathcal{L}$; and
2. $afc_\ell(\lambda) = \emptyset$ for all $\lambda \in \mathcal{S}$.

In the following example, we define a $\mathcal{C}(Pet)$-assignment for our specification of Peterson's algorithm.

Example 23 Recall that the set of components $\mathcal{C}(Pet)$ associated with *Pet* is

$$\mathcal{C}(Pet) = \{\text{L}, \ \text{RL}, \ \text{RRL}, \ \text{RRRL}, \ \text{RRRR}\} \ .$$

To define the mappings $npc_\ell, afc_\ell : \mathcal{L} \to 2^{\mathcal{C}(Pet)}$ it is convenient to first associate with every component $\sigma \in \mathcal{C}(Pet)$ a set of labels $\mathcal{L}_\sigma \subseteq \mathcal{L}$. We have

$$\mathcal{L}_{\text{L}} = \{\textbf{noncritA}, s\_asgn^{true}_{RA}, s\_asgn^{B}_{T}, r\_rd^{false}_{RB}, r\_rd^{A}_{T}, \textbf{critA}, s\_asgn^{false}_{RA}\},$$
$$\mathcal{L}_{\text{RL}} = \{\textbf{noncritB}, s\_asgn^{true}_{RB}, s\_asgn^{A}_{T}, r\_rd^{false}_{RA}, r\_rd^{B}_{T}, \textbf{critB}, s\_asgn^{false}_{RB}\},$$

$$\mathcal{L}_{\text{RRL}} = \{r\_asgn_{RA}^{true}, r\_asgn_{RA}^{false}, r\_rd_{RA}^{true}, r\_rd_{RA}^{false}\},$$
$$\mathcal{L}_{\text{RRRL}} = \{r\_asgn_{RB}^{true}, r\_asgn_{RB}^{false}, r\_rd_{RB}^{true}, r\_rd_{RB}^{false}\}, \text{ and}$$
$$\mathcal{L}_{\text{RRRR}} = \{r\_asgn_{T}^{A}, r\_asgn_{T}^{B}, r\_rd_{T}^{A}, r\_rd_{T}^{B}\}.$$

Now we can define, for all $\sigma \in \mathcal{C}(Pet)$ and all $\lambda \in \mathcal{L}_\sigma$:

$$npc_\ell(\lambda) = \{\sigma\}, \quad \text{and} \quad afc_\ell(\lambda) = \begin{cases} \{\sigma\} & \text{if } \lambda \in \mathcal{A}, \text{ and} \\ \emptyset & \text{if } \lambda \in \mathcal{S} \ . \end{cases}$$

On the other elements of $\mathcal{L}$, the results of communications, npc_ℓ and afc_ℓ are defined as follows:

$$\begin{aligned}
&npc_\ell(asgn_{RA}^b) = afc_\ell(asgn_{RA}^b) = \{\text{L, RRL}\} && (b \in \{true, false\}),\\
&npc_\ell(asgn_{RB}^b) = afc_\ell(asgn_{RB}^b) = \{\text{RL, RRRL}\} && (b \in \{true, false\}),\\
&npc_\ell(asgn_{T}^B) = afc_\ell(asgn_{T}^B) = \{\text{L, RRRR}\},\\
&npc_\ell(asgn_{T}^A) = afc_\ell(asgn_{T}^A) = \{\text{RL, RRRR}\},\\
&npc_\ell(rd_{RA}^b) = \{\text{RL, RRL}\}, \quad afc_\ell(rd_{RA}^b) = \{\text{RL}\} && (b \in \{true, false\}),\\
&npc_\ell(rd_{RB}^b) = \{\text{L, RRRL}\}, \quad afc_\ell(rd_{RB}^b) = \{\text{L}\} && (b \in \{true, false\}),\\
&npc_\ell(rd_{T}^A) = \{\text{L, RRRR}\}, \quad afc_\ell(rd_{T}^A) = \{\text{L}\}, \text{ and}\\
&npc_\ell(rd_{T}^B) = \{\text{RL, RRRR}\}, \quad afc_\ell(rd_{T}^B) = \{\text{RL}\}.
\end{aligned}$$

It is easy to verify that (npc_ℓ, afc_ℓ) satisfies the requirements of Definition 22 and hence is a $\mathcal{C}(Pet)$-assignment.

We could now proceed to prove directly that the $\mathcal{C}(Pet)$-assignment in the preceding example satisfies Eqs. (4) and (5) and conclude that the LTSC associated with *Pet* has a concurrency-consistent labelling. We prefer to proceed more generally, however, and define a subclass of process expressions together with assumptions on the underlying recursive specification that guarantee that an assignment satisfying Eqs. (4) and (5) exists. It will be easy to verify that *Pet* is a process expression in the subclass, and that the recursive specification E_{Pet} satisfies the assumptions, from which it will follow that the $\mathcal{C}(Pet)$-assignment above indeed satisfies Eqs. (4) and (5). In fact, it can be checked automatically whether a process expression is in the subclass and the underlying recursive specification satisfies the assumptions.

We consider parallel compositions of sequential components. These sequential components should have disjoint alphabets and respect the use of signals. Moreover, the communication function should support a consistent assignment of components to labels. Below, we shall first formulate sufficient conditions on a sequential process expression and its underlying sequential recursive specification that ensure that transitions labelled with signals do not change state in the LTSC associated with the process expression. Then, we associate with every sequential process expression its (reachable) alphabet and its (reachable) action alphabet, so that we can formulate the requirement that the alphabets of components are disjoint. And finally we shall define when an assignment is consistent with a communication function.

Sequential components The set of *sequential process expressions* is generated by the following grammar (with A ranging over $\mathcal{I}$ and λ ranging over $\mathcal{L}$):

$$S ::= \mathbf{0} \mid \lambda.S \mid S + S \mid A \ .$$

By a *sequential recursive specification* E we mean a set of defining equations

$$A \stackrel{\text{def}}{=} S_A \ ,$$

with S_A a sequential process expression, including precisely one such equation for every $A \in \mathcal{I}$.

A sequential process expression S is *syntactically guarded* if all occurrences of process identifiers in S are within the scope of an action prefix. A sequential recursive specification E is *syntactically guarded* if for every defining equation $A \stackrel{\text{def}}{=} S_A$ in E it holds that S_A is syntactically guarded.

Respect for signals Let E be a sequential recursive specification, and let us denote, for all $A \in \mathcal{I}$, by S_A the right-hand side of the defining equation for A in E. We say that $A \in \mathcal{I}$ is *signalling* if S_A has a subexpression $\lambda.A$ with $\lambda \in \mathcal{S}$. A process identifier $A \in \mathcal{I}$ is *signal-respecting* if

1. for every subexpression $\lambda.S'$ of S_A with $\lambda \in \mathcal{S}$ it holds that $S' = A$ and the occurrence of the subexpression is not in the scope of another prefix, and
2. for every subexpression $S_1 + S_2$ of S_A it holds that S_1 and S_2 are not signalling process identifiers.

E is *signal-respecting* if it is syntactically guarded and all process identifiers in $\mathcal{I}$ are signal-respecting. A sequential process expression S is *signal-respecting* with respect to a signal-respecting sequential recursive specification E if S does not have subexpressions of the form $\lambda.S'$ with $\lambda \in \mathcal{S}$, and for every subexpression $S_1 + S_2$ it holds that S_1 and S_2 are not signalling process identifiers.

Example 24 It is straightforward to check that E_{Pet} is a syntactically guarded sequential recursive specification and that it is signal-respecting.

Lemma 25 *Let E be a signal-respecting recursive specification and let t be a transition such that $src(t)$ is a signal-respecting sequential process expression. Then $target(t)$ is again a signal-respecting sequential process expression, and t is a signal transition if, and only if, $\ell(t) \in \mathcal{S}$.*

Proof To establish that $target(t)$ is again a signal-respecting sequential process expression, we first note that if $A \stackrel{\text{def}}{=} S_A$ is the equation in E defining some process identifier A, and $S_A \xrightarrow{\lambda,\alpha,\varsigma} S'$, then S' is signal-respecting. For by syntactic guardedness, S' is a subexpression of S_A, by the first requirement satisfied by signal-respecting process identifiers S' cannot have subexpressions of the form $\lambda.S''$ with $\lambda \in \mathcal{S}$, and by the second requirement satisfied by signal-respecting process identifiers, whenever $S_1 + S_2$ is a subexpression of S', then S_1 and S_2 cannot be signalling process identifiers. We can now argue that $target(t)$ is a signal-respecting sequential process expression with a straightforward induction on the structure of $src(t)$.

It remains to show that t is a signal transition if, and only if, $\ell(t) \in \mathcal{S}$.

For the implication from left to right, note that if t is a signal transition, then, by definition, $\ell(t) \in \mathcal{S}$.

For the converse implication, suppose that $\ell(t) \in \mathcal{S}$; we need to establish that $src(t) = target(t)$. To this end, we first establish with induction on the structure of S that if S is a

signal-respecting process expression, $\lambda \in \mathcal{S}$ and $S \xrightarrow{\lambda,\alpha,\varsigma} S'$, then $S = A$ for some process identifier A. Clearly, S cannot be $\mathbf{0}$. Furthermore, since signal-respecting sequential process expressions do not have subexpressions of the form $\lambda.S''$ with $\lambda \in \mathcal{S}$, we cannot have that $S = \lambda.S''$ for some process expression S''. Note that if we would have $S = S_1 + S_2$, then either $S_1 \xrightarrow{\lambda,\alpha,\varsigma} S'$ or $S_2 \xrightarrow{\lambda,\alpha,\varsigma} S'$, so by the induction hypothesis either S_1 or S_2 would be a signalling process identifier, contradicting the assumption that for every subexpression $S_1 + S_2$ of S it holds that S_1 and S_2 are not signalling process identifiers. It follows that $S = A$ for some (signalling) process identifier A. Hence, assuming that $(A \stackrel{\text{def}}{=} S_A) \in E$, t has a subderivation t' with $src(t') = S_A$ and $\ell(t') \in \mathcal{S}$. From the first requirement satisfied by signal-respecting process identifiers it now follows that $target(t) = target(t') = A$. □

Alphabet We also wish to associate with each sequential process expression S its *alphabet* $\mathcal{L}(S)$ and its *action alphabet* $\mathcal{A}(S)$, the sets of labels of transitions and action transitions reachable from S, respectively. To this end, we first define $\mathcal{L}(A)$ for all process identifiers defined in E, using two auxiliary notions. First, we associate with every sequential process expression S its non-recursive alphabet $\mathcal{L}'(S)$ inductively by: $\mathcal{L}'(\mathbf{0}) = \emptyset$, $\mathcal{L}'(A) = \emptyset$ for all $A \in \mathcal{I}$, $\mathcal{L}'(\lambda.S) = \{\lambda\} \cup \mathcal{L}'(S)$, and $\mathcal{L}'(S_1 + S_2) = \mathcal{L}'(S_1) \cup \mathcal{L}'(S_2)$. Second, we define on $\mathcal{I}$ a binary relation $\triangleright$ by $A \triangleright A'$ if $A \stackrel{\text{def}}{=} S$ in E and A' occurs in S, and denote by $\triangleright^*$ the reflexive-transitive closure of $\triangleright$. Then we can define the alphabet $\mathcal{L}(A)$ of A by

$$\mathcal{L}(A) = \bigcup \{\mathcal{L}'(S) \mid A \triangleright^* A' \text{ and } A' \stackrel{\text{def}}{=} S\} \ .$$

Now, we inductively extend $\mathcal{L}(\_)$ to all sequential process expressions defining $\mathcal{L}(\mathbf{0}) = \emptyset$, $\mathcal{L}(\lambda.S) = \{\lambda\} \cup \mathcal{L}(S)$, and $\mathcal{L}(S_1 + S_2) = \mathcal{L}(S_1) \cup \mathcal{L}(S_2)$. Furthermore, we define $\mathcal{A}(S) = \mathcal{L}(S) \cap \mathcal{A}$.

Lemma 26 *Let E be a sequential recursive specification and let S be a sequential process expression over E. If S' is a sequential process expression reachable from S, then $\mathcal{L}(S') \subseteq \mathcal{L}(S)$ and $\mathcal{A}(S') \subseteq \mathcal{A}(S)$.*

Proof We first consider the special case that there is a transition t with $src(t) = S$ and $target(t) = S'$ and prove with induction on t that $\mathcal{L}(S') \subseteq \mathcal{L}(S)$.

If the last rule applied in t is (PREF), then we have $S = \lambda.S'$ and hence $\mathcal{L}(S') \subseteq \{\lambda\} \cup \mathcal{L}(S') = \mathcal{L}(S)$.

If the last rule applied in t is (SUM- L), then there exist S_1 and S_2 such that $S = S_1 + S_2$, and t has a subderivation t' with $src(t') = S_1$ and $target(t') = S'$. By the induction hypothesis we have that $\mathcal{L}(S') \subseteq \mathcal{L}(S_1) \subseteq \mathcal{L}(S_1) \cup \mathcal{L}(S_2) = \mathcal{L}(S)$.

If the last rule applied in t is (SUM- R), then there exist S_1 and S_2 such that $S = S_1 + S_2$, and t has a subderivation t' with $src(t') = S_2$ and $target(t') = S'$. By the induction hypothesis we have that $\mathcal{L}(S') \subseteq \mathcal{L}(S_2) \subseteq \mathcal{L}(S_1) \cup \mathcal{L}(S_2) = \mathcal{L}(S)$.

If the last rule applied in t is (REC), then $S = A$ for some process identifier $A \in \mathcal{I}$ with defining equation $(A \stackrel{\text{def}}{=} S_A) \in E$, and t has a subderivation t' with $src(t') = S_A$ and $target(t') = S'$. By the induction hypothesis, $\mathcal{L}(S') \subseteq \mathcal{L}(S_A)$; it therefore remains to show that $\mathcal{L}(S_A) \subseteq \mathcal{L}(A)$. We have:

$$\begin{aligned} \mathcal{L}(S_A) &= \mathcal{L}'(S_A) \cup \bigcup \{\mathcal{L}(A') \mid A \triangleright A'\} \\ &= \mathcal{L}'(S_A) \cup \bigcup \{\mathcal{L}'(S_{A''}) \mid A \triangleright A' \triangleright^* A''\} = \bigcup \{\mathcal{L}'(S_{A'}) \mid A \triangleright^* A'\} = \mathcal{L}(A) \ . \end{aligned}$$

[In the second equality we have used that $A \triangleright^* A''$ for all A' such that $A \triangleright A'$. In the third equality we have used the definition of $\mathcal{L}(A)$.]

Now, if S' is reachable from S, then the statement of the lemma follows with a straightforward induction on the number of transitions in a path from S to S'. Furthermore, it is then immediate from the definition of action alphabet that $\mathcal{A}(S') \subseteq \mathcal{A}(S)$. □

Parallel-sequential processes Presupposing a signal-respecting sequential recursive specification E, a *parallel-sequential* process expression over E is a process expression generated by the following grammar (with S ranging over sequential process expressions and $H \subseteq \mathcal{L}$):

$$P ::= S \mid P \parallel P \mid \partial_H(P) \ .$$

Lemma 27 *Let E be a sequential recursive specification and let P be a parallel-sequential process expression over E. If P' is reachable from P, then $\mathcal{C}(P') = \mathcal{C}(P)$ and $P'|_\sigma$ is reachable from $P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.*

Proof With induction on t it can be established that if t is a transition such that $src(t) = P$ and $target(t) = P'$, then $\mathcal{C}(P') = \mathcal{C}(P)$ and $P'|_\sigma = P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$. The details are worked out in "Appendix B" (see Lemma 47).

Then, if P' is reachable from P, the statement of the lemma follows with a straightforward induction on the number of transitions in a path from P to P'. □

Since a communication function γ is required to be commutative and associative, it induces a partial function $\overline{\gamma} : \mathcal{M}_f(\mathcal{L}) \rightharpoonup \mathcal{L}$, where $\mathcal{M}_f(\mathcal{L})$ denotes the set of all finite multisets over $\mathcal{L}$. We define $\overline{\gamma}([\lambda_0, \ldots, \lambda_n])$ with induction on n as follows:

1. If $n = 0$, then $\overline{\gamma}([\lambda_0, \ldots, \lambda_n]) = \lambda_0$.
2. If $n = 1$, then $\overline{\gamma}([\lambda_0, \ldots, \lambda_n]) = \gamma(\lambda_0, \lambda_n)$ if $\gamma(\lambda_0, \lambda_n)$ is defined, and undefined otherwise.
3. If $n \geq 1$, then $\overline{\gamma}([\lambda_0, \ldots, \lambda_{n+1}]) = \gamma(\overline{\gamma}([\lambda_0, \ldots, \lambda_n]), \lambda_{n+1})$ if both $\overline{\gamma}([\lambda_0, \ldots, \lambda_n])$ and $\gamma(\overline{\gamma}([\lambda_0, \ldots, \lambda_n]), \lambda_{n+1})$ are defined, and undefined otherwise.

It is straightforward to prove, with induction on $n \geq 1$, that for all $\lambda_0, \ldots, \lambda_n$ and for all $0 \leq k < n$ that $\gamma(\overline{\gamma}(\lambda_0, \ldots, \lambda_k), \overline{\gamma}(\lambda_{k+1}, \ldots, \lambda_n)) = \overline{\gamma}(\lambda_0, \ldots, \lambda_n)$; we shall use this fact in the proof of the next lemma, which relates transitions of a parallel-sequential process with transitions of its components.

Lemma 28 *Let t be a transition, let $npc(t) = \{\sigma_0, \ldots, \sigma_n\}$, and suppose that $src(t)$ is a parallel-sequential process expression. Then t has subderivations $t_0, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression and $src(t_i) = src(t)|_{\sigma_i}$ for all $0 \leq i \leq n$, and $\ell(t) = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$ (where $[\ell(t_0), \ldots, \ell(t_n)]$ denotes the multiset over $\mathcal{L}$ consisting of $\ell(t_0), \ldots, \ell(t_n)$).*

Proof We proceed by induction on t.

If the last rule applied in t is (PREF), (SUM-L), (SUM-R), or (REC), then $npc(t) = \{\epsilon\}$, $src(t) = src(t)|_\epsilon$, and $\ell(t) = \overline{\gamma}([\ell(t)])$. Moreover, from the syntax definition of parallel-sequential processes it is clear that $src(t)$ is a sequential process expression.

If the last rule applied in t is (PAR-L), then t has a subderivation t' such that $npc(t) = \mathrm{L} \rhd npc(t')$, so there exist static components $\sigma'_0, \ldots, \sigma'_n$ such that $\sigma_i = \mathrm{L}\sigma'_i$ for all $0 \leq i \leq n$. By the induction hypothesis, t', and hence t, has subderivations $t_0, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression, $src(t_i) = src(t')|_{\sigma'_i} = src(t)|_\sigma$ for all $0 \leq i \leq n$ and $\ell(t) = \ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$.

If the last rule applied in t is (PAR-R), then the proof proceeds analogously.

If the last rule applied in t is (COMM), then t has subderivations t' and t'' such that $npc(t) = \mathrm{L} \rhd npc(t') \cup \mathrm{R} \rhd npc(t'')$. Since $npc(t')$ and $npc(t'')$ cannot be empty, we have that $n \geq 1$ and there exist static components $\sigma'_0, \ldots, \sigma'_n$ and a $0 \leq k < n$ such that $npc(t') = \{\sigma'_0, \ldots, \sigma'_k\}$ and $npc(t'') = \{\sigma'_{k+1}, \ldots, \sigma_n\}$. By the induction hypothesis, t' and t'', and hence t, have subderivations $t_0, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression for all $0 \leq i \leq n$, $src(t_i) = src(t')|_{\sigma'_i} = src(t)|_{\sigma_i}$ for all $0 \leq i \leq k$, $\ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_k)])$, $src(t_i) = src(t'')|_{\sigma'_i} = src(t)|_{\sigma_i}$ for all $k < i \leq n$, and $\ell(t'') = \overline{\gamma}([\ell(t_{k+1}), \ldots, \ell(t_n)])$. Furthermore, we have that

$$\begin{aligned}\ell(t) &= \gamma(\ell(t'), \ell(t'')) = \gamma(\overline{\gamma}([\ell(t_0), \ldots, \ell(t_k)]), \overline{\gamma}([\ell(t_{k+1}), \ldots, \ell(t_n)])) \\ &= \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)]) \ .\end{aligned}$$

Finally, if the last rule applied in t is (ENC), then t has a subderivation t' with $npc(t') = \{\sigma_0, \ldots, \sigma_n\}$ and $\ell(t') = \ell(t)$, so it follows immediately by the induction hypothesis that there exist subderivations $t_0, \ldots, t_n$ of t' and hence of t such that $src(t_i) = src(t')|_{\sigma_i} = src(t)|_{\sigma_i}$ and $\ell(t) = \ell(t') = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$. □

Definition 29 Let $C \subseteq \mathcal{C}^*$ be a finite set of static components. A C-assignment (npc_ℓ, afc_ℓ) is *consistent* with a communication function γ if it satisfies, for all $\lambda_1, \lambda_2, \lambda_3 \in \mathcal{L}$ such that $\gamma(\lambda_1, \lambda_2) = \lambda_3$:

1. $npc_\ell(\lambda_1) \cup npc_\ell(\lambda_2) = npc_\ell(\lambda_3)$; and
2. $afc_\ell(\lambda_1) \cup afc_\ell(\lambda_2) = afc_\ell(\lambda_3)$.

Example 30 Consider the specification of Peterson's algorithm, it is straightforward to verify that the $\mathcal{C}(Pet)$-assignment (npc_ℓ, afc_ℓ) presented in Example 23 is consistent with the communication function γ. Consider, by way of example, the equation

$$\gamma(r\_asgn^{true}_{RA}, s\_asgn^{true}_{RA}) = asgn^{true}_{RA} \ ,$$

which is part of the definition of γ. We confirm as follows that indeed the conditions of Definition 29 are satisfied:

$$\begin{aligned}npc_\ell(r\_asgn^{true}_{RA}) \cup npc_\ell(s\_asgn^{true}_{RA}) &= \{\mathrm{L}, \mathrm{RRL}\} = npc_\ell(asgn^{true}_{RA}), \\ afc_\ell(r\_asgn^{true}_{RA}) \cup afc_\ell(s\_asgn^{true}_{RA}) &= \{\mathrm{L}, \mathrm{RRL}\} = afc_\ell(asgn^{true}_{RA}).\end{aligned}$$

If $npc_\ell : \mathcal{L} \to 2^C$ associates with every label a subset of components in C and $C' \subseteq C$, then we denote by $\mathcal{L}(C')$ the *alphabet* of C', i.e.,

$$\mathcal{L}(C') = npc_\ell^{-1}(C') = \{\lambda \in \mathcal{L} \mid npc_\ell(\lambda) = C'\},$$

and by $\mathcal{A}(C')$ the *action alphabet* of C', i.e.,

$$\mathcal{A}(C') = afc_\ell^{-1}(C') = \{\lambda \in \mathcal{L} \mid afc_\ell(\lambda) = C'\}.$$

Note that by condition 2 of Definition 22 we have $\mathcal{A}(C') \subseteq \mathcal{A}$.

Theorem 31 *Let E be a signal-respecting sequential recursive specification, let P be a parallel-sequential process expression over E, and let (npc_ℓ, afc_ℓ) be a $\mathcal{C}(P)$-assignment. If $\mathcal{L}(P|_\sigma) \subseteq \mathcal{L}(\{\sigma\})$ and $\mathcal{A}(P|_\sigma) \subseteq \mathcal{A}(\{\sigma\})$ for all $\sigma \in \mathcal{C}(P)$ and γ is signal-respecting and consistent with (npc_ℓ, afc_ℓ), then (npc_ℓ, afc_ℓ) satisfies the requirements (4) and (5) for every transition t reachable from P.*

Proof Let t be a transition reachable from P. Then $src(t) = P'$ for some parallel-sequential process expression P' reachable from P. By Lemma 27 we have $\mathcal{C}(P') = \mathcal{C}(P)$ and we have $P'|_c$ is reachable from $P|_c$ for all $c \in \mathcal{C}(P)$. So, without loss of generality, we may assume that $src(t) = P$.

Let $npc(t) = \{\sigma_0, \ldots, \sigma_n\}$. By Lemma 28, t has subderivations $t_1, \ldots, t_n$ such that $src(t_i)$ is a sequential process expression and $src(t_i) = src(t)|_{\sigma_i}$ for $0 \leq i \leq n$, and $\ell(t) = \overline{\gamma}([\ell(t_0), \ldots, \ell(t_n)])$. Since, for all $0 \leq i \leq n$, $\ell(t_i) \in \mathcal{L}(P|_{\sigma_i}) \subseteq \mathcal{L}(\{\sigma_i\})$, we have $npc_\ell(\ell(t_i)) = \{\sigma_i\}$, and hence, by condition 1 of Definition 29,

$$npc(t) = \{\sigma_i \mid 0 \leq i \leq n\} = \bigcup_{0 \leq i \leq n} npc_\ell(\ell(t_i)) = npc_\ell(\ell(t)) \ .$$

Since E is a signal-respecting recursive specification and $src(t_i)$ is a signal-respecting sequential process expression, by Lemma 25 t_i is a signal transition if, and only if, $\ell(t_i) \in \mathcal{S}$. Since, on the one hand, $afc_\ell(\ell(t_i)) = \emptyset$ for all $\ell(t_i) \in \mathcal{S}$, and, on the other hand, $\mathcal{L}(P|_\sigma) \cap \mathcal{A} \subseteq \mathcal{A}(\{\sigma\})$ we have

$$\begin{aligned} afc(t) &= \{\sigma_i \mid 0 \leq i \leq n \ \&\ \lambda_i \in \mathcal{A}\} \\ &= \bigcup \{afc_\ell(\ell(t_i)) \mid 0 \leq i \leq n \ \&\ \ell(t_i) \in \mathcal{A}\} = afc_\ell(\ell(t)) \ . \end{aligned}$$

This completes the proof of the theorem. □

If E, P, γ and (npc_ℓ, afc_ℓ) satisfy the requirements of the preceding theorem, then the relation $\rightsquigarrow$ on labels by $\lambda_1 \rightsquigarrow \lambda_2$ if, and only if, $npc_\ell(\lambda_1) \cap afc_\ell(\lambda_2) = \emptyset$ satisfies the requirements of Definition 18. So we get the following corollary.

Corollary 32 *Let E be a signal-respecting sequential recursive specification, let P be a parallel-sequential process expression over E, and let (npc_ℓ, afc_ℓ) be a $\mathcal{C}(P)$-assignment such that $\mathcal{L}(P|_\sigma) \subseteq \mathcal{L}(\{\sigma\})$ and $\mathcal{A}(P|_\alpha) \subseteq \mathcal{A}(\{\sigma\})$ for all $\sigma \in \mathcal{C}(P)$ and γ is signal-respecting and consistent with (npc_ℓ, afc_ℓ). Then the LTSC associated with P has a concurrency-consistent labelling.*

Example 33 In Example 30 we have established that all the conditions of Corollary 32 are satisfied for E_{Pet}, γ, *Pet* and the $\mathcal{C}(Pet)$-assignment (npc_ℓ, afc_ℓ) defined in Example 23, so the LTSC associated with *Pet* has a concurrency-consistent labelling.

8 Expressing liveness

A mathematically rigorous method for establishing the correctness of a (finite model of a) system is by means of *model checking*. Given a process expression specifying a system, the behaviours of that system can be scrutinised by verifying which requirements, expressed in a modal logic, hold true and which ones fail to hold. Among the modal logics that can be

used to express such requirements is the modal μ-calculus. This is one of the most expressive logics available, subsuming logics such as LTL, CTL and CTL*, and it is typically used in toolset for analysing labelled transition systems, such as the mCRL2 toolset [2] and CADP [6]. We introduce this logic in Sect. 8.1.

Liveness requirements typically assert that (conditionally or unconditionally) something good must inevitably happen. Phrasing such properties in the modal μ-calculus is rather standard, but it is less clear whether the logic permits expressing liveness properties restricted to just paths only. This is partly due to the fact that justness is a predicate on paths, whereas the modal μ-calculus is a state-based formalism, and partly due to the 'dynamic' nature of justness, which checks along a path for enabledness of actions and their future elimination. In particular this dynamic nature rules out a 'static' encoding such as the one presented in [5] for dealing with fairness, as it assumes an *a priori* fixed—i.e., static—collection of constraints that need to hold infinitely often for a path to be fair.

We show that liveness requirements of the form 'along every just path, every a action is inevitably followed by a b action' can indeed be expressed in the modal μ-calculus. Other path-based properties can be defined along the same lines. We discuss the liveness property in Sect. 8.2.

8.1 The modal μ-calculus

The modal μ-calculus can be viewed as a fixed point extension of *Hennessy–Milner Logic* (HML) [13]. In HML one can characterise the capabilities of a state to execute actions using modal operators $[\_]\_$ and $\langle\_\rangle\_$; essentially, this permits to reason about the transitions emanating from a state. Fixed points add the power of recursion to these basic facilities; this, intuitively, allows to reason about finite or infinite sequences or trees of transitions and the capabilities of the states visited along such sequences or trees. The resulting logic, i.e., HML with fixed points, is referred to as the modal μ-calculus (L_μ). For an in-depth treatment of this logic, we refer to, e.g. [14].

Our syntax of the modal μ-calculus is given in the context of a set of recursion variables $\mathcal{V}$, in addition to a finite set of labels $\mathcal{L}$. The set Φ of formulas of L_μ is generated by the following grammar (with X ranging over the set of variables $\mathcal{V}$, and λ ranging over the finite set of labels $\mathcal{L}$):

$$\varphi ::= X \mid \top \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\lambda\rangle\varphi \mid [\lambda]\varphi \mid \mu X.\varphi \mid \nu X.\varphi \ .$$

The binding precedence of the operators is as usual, with the fixed point operators binding weakest. We permit ourselves to write $\bigwedge_{\lambda\in A}\phi(\lambda)$ and $\bigvee_{\lambda\in A}\phi(\lambda)$ for a set of actions A, as generalisations of the binary conjunction and disjunction.

Let $(St, Tr, src, target, \ell, \frown\!\!\bullet)$ be a finite LTSC over $\mathcal{L}$ with a concurrency-consistent labelling. We proceed to give a denotational semantics for our logic by associating every formula φ with the subset $[\![\varphi]\!]_\vartheta \subseteq St$ of states in which it holds; since formulas may contain free variables, $[\![\varphi]\!]_\vartheta$ is relative to an assignment ϑ that provides an interpretation of recursion variables $X \in \mathcal{V}$ as subsets of St. We define $[\![\cdot]\!]_\vartheta$ recursively as follows:

$$\begin{aligned}
[\![X]\!]_\vartheta &= \vartheta(X);\\
[\![\top]\!]_\vartheta &= St;\\
[\![\bot]\!]_\vartheta &= \emptyset;\\
[\![\varphi \wedge \psi]\!]_\vartheta &= [\![\varphi]\!]_\vartheta \cap [\![\psi]\!]_\vartheta;\\
[\![\varphi \vee \psi]\!]_\vartheta &= [\![\varphi]\!]_\vartheta \cup [\![\psi]\!]_\vartheta;
\end{aligned}$$

$$\begin{aligned}
[\![\langle\lambda\rangle\varphi]\!]_\vartheta &= \{s \in St \mid \exists t \in Tr.\ s = src(t)\ \&\ \ell(t) = \lambda\ \&\ target(t) \in [\![\varphi]\!]_\vartheta\};\\
[\![[\lambda]\varphi]\!]_\vartheta &= \{s \in St \mid \forall t \in Tr.\ (s = src(t)\ \&\ \ell(t) = \lambda) \rightarrow target(t) \in [\![\varphi]\!]_\vartheta\};\\
[\![\mu X.\,\varphi]\!]_\vartheta &= \bigcap\{\mathcal{F} \subseteq St \mid [\![\varphi]\!]_{\vartheta[X:=\mathcal{F}]} \subseteq \mathcal{F})\};\ \text{and}\\
[\![\nu X.\,\varphi]\!]_\vartheta &= \bigcup\{\mathcal{F} \subseteq St \mid \mathcal{F} \subseteq [\![\varphi]\!]_{\vartheta[X:=\mathcal{F}]})\}.
\end{aligned}$$

Note that the structure $(2^{St}, \subseteq)$ is a complete lattice. For an endofunction $\mathcal{T} : 2^{St} \rightarrow 2^{St}$, we write $\mu\mathcal{F}.\,\mathcal{T}(\mathcal{F})$ and $\nu\mathcal{F}.\,\mathcal{T}(\mathcal{F})$ to denote the least and greatest fixed points of $\mathcal{T}$, respectively. The interpretation of a formula φ is *independent* of the valuation ϑ in case it contains no unbound recursion variables (i.e., all occurrences of a recursion variable are within the scope of a least or greatest fixed point). We simply write $[\![\varphi]\!]$ when referring to the semantics of such a formula, as it yields the same set of states for every possible environment ϑ used to interpret φ.

Example 34 Greatest fixed point formulas typically characterise invariant properties, whereas least fixed point formulas characterise liveness properties. For instance, the L_μ formula $\nu X.\,\langle a\rangle X \wedge [b]\bot$, asserts the existence of an infinite a-path along which no b-action can be executed; this is an invariant property along the path. On the other hand, the formula $\mu X.\,\langle a\rangle X \vee \langle b\rangle\top$ asserts that there is a finite path of a-labelled transitions, leading to a state in which a b-labelled transition is enabled.

8.2 Expressing liveness along just paths

We consider liveness properties of the kind 'whenever some non-blocking action **a** happens, then inevitably also **b** happens'; this property will be referred to as **a**-**b**-*liveness* and a state is said to satisfy **a**-**b**-liveness exactly when all paths emanating from that state satisfy **a**-**b**-liveness. An L_μ formula that asserts that this property holds along all paths in a given (deadlock-free) LTS is the following

$$\nu X.\left(\bigwedge_{\lambda\in\mathcal{A}}[\lambda]X \wedge [\mathbf{a}]\mu Y.\bigwedge_{\lambda\in\mathcal{A}\setminus\{\mathbf{b}\}}[\lambda]Y\right).$$

Restricting **a**-**b**-liveness to *just* paths requires that somehow the concept of justness is woven into this formula. We explain in several steps how this can be achieved.

In order to facilitate our reasoning, we consider the dual problem of characterising an **a**-**b**-liveness violation along *some* just path. While this problem is technically equally difficult, it is conceptually simpler since we are now only concerned with constructing a formula that describes the *existence* of a just path. Notice that a just path constitutes a violation to **a**-**b**-liveness precisely when (1) this path has a suffix starting at a state s', reached by an **a**-labelled transition, along which action **b** never takes place and (2) the path is just.

Our approach to characterising states that admit a violating path (should one exist) is based on the following observation. In our setting, any just path can be prefixed by an arbitrary finite path, resulting in a new just path (see Proposition 35 below). This means that we can characterise states that admit a just, **b**-free path. Given any such state, we can characterise the states reaching it via a path ending with an **a**-labelled transition.

For the remainder of this section we fix a finite LTSC $(St, Tr, src, target, \ell, \smile)$ with a concurrency-consistent labelling. The justness rephrasing of Proposition 19 requires one to reason about the enabled actions of a state. Let $\mathsf{En}(s)$ be the set of enabled non-blocking actions: $\mathsf{En}(s) = \{\lambda \in \overline{\mathcal{B}} \mid \exists t \in Tr : src(t) = s\ \&\ \ell(t) = \lambda\}$.

Table 3 Template formula that characterises the set of states that admit a just path violating **a**-**b**-liveness

$$\mathsf{violate} = \mu W.\ \big(\langle \mathbf{a}\rangle \mathsf{invariant} \vee \bigvee_{\lambda\in\mathcal{A}} \langle\lambda\rangle W\big)$$
$$\mathsf{invariant} = \nu Y.\ \bigwedge_{\lambda\in\overline{\mathcal{B}}} (\langle\lambda\rangle\top \Rightarrow \mathsf{elim}(\lambda))$$
$$\mathsf{elim}(\lambda) = \mu Q.\ \Big(\bigvee_{\lambda'\in \#\lambda\setminus\{\mathbf{b}\}} \langle\lambda'\rangle Y \vee \bigvee_{\lambda'\in\mathcal{A}\setminus(\#\lambda\cup\{\mathbf{b}\})} \langle\lambda'\rangle Q\Big)$$
$$\text{where } \#\lambda = \{\lambda' \mid \lambda \smile\!\!\!\!\!\!\bullet\ \lambda'\}$$

Subformula invariant characterises the set of states that admit a just, **b**-free path. The user provides the sets $\mathcal{A}$ and $\overline{\mathcal{B}}$, the relation $\smile\!\!\!\!\!\!\bullet$ and the pair of actions **a** and **b** to instantiate/generate the formula for checking a concrete LTSC

Proposition 35 *Let π be a $\mathcal{B}$-just path. Then the path $s_0t_1s_1\ldots t_n\pi$ is $\mathcal{B}$-just.*

Proof Let $\pi' = s_0t_1s_1t_2\ldots t_n\pi$ be a path such that π is $\mathcal{B}$-just, and let s_π be the starting state of π. Suppose s is a state on π' and $\lambda \in \mathsf{En}(s)$. We distinguish two cases.

- Case s does not occur in the prefix $s_0t_1s_1t_2\ldots t_n$. Then s occurs in π and since π is $\mathcal{B}$-just, λ is eliminated in the suffix of π (and therefore also in the suffix of π'), starting in s.
- Case s occurs in the prefix $s_0t_1s_1t_2\ldots t_n$. Towards a contradiction, assume that λ is not eliminated in the suffix of π' starting in s. Let t be the transition such that $\ell(t) = \lambda$ and $src(t) = s$. Since λ is not eliminated in the suffix of π' starting in s and s_π is reachable from s, by condition 2 of Definition 2, there must be an action transition u such that $src(u) = s_\pi$ and $\ell(t) = \lambda = \ell(u)$. But then $\lambda \in \mathsf{En}(s_\pi)$ and, since π is $\mathcal{B}$-just, λ is eliminated in π. Contradiction. Consequently, λ is eliminated in the suffix of π' starting in s. □

The suffixes of a just path are again just. This is formalised by the following proposition.

Proposition 36 *Let $\pi = s_0t_1s_1t_2\ldots$ be a finite or infinite path. If π is $\mathcal{B}$-just then also any suffix of π is $\mathcal{B}$-just.*

Proof Let π be a $\mathcal{B}$-just path and let π' be a suffix of π. Pick some state s in π' and an action $\lambda \in \mathsf{En}(s)$. Since s is in π', s is also in π. Consequently, λ must be eliminated by some action in the suffix of π starting at s. Since s is in π', the suffix of π starting at s also is a suffix of π'. □

We next lift the notion of just path to the level of states: a *state* is just whenever it is the start of a just path. Note that we are interested in characterising states that admit a just path that constitute an **a**-**b**-liveness violation; such paths must have suffixes that are void of **b**-actions. For this reason, we parameterise the notion of a just state with a set of actions K that limits the set of actions allowed to occur along the just paths.

Definition 37 Let $K \subseteq \mathcal{L}$ be a non-empty set of actions. We define $\mathcal{J}(K)$ as follows:

$$\mathcal{J}(K) = \{s \in St \mid \text{there is a } \mathcal{B}\text{-just path } \pi \text{ starting in } s \text{ and } \ell(t) \in K \text{ for all transitions } t \text{ on } \pi\}$$

As we explained at the beginning of this section, we tackle our problem in two steps. First we show that formula invariant, see Table 3, characterises the states that admit a just path along which no **b**-action ever happens; i.e., those are essentially the states in the set $\mathcal{J}(\mathcal{A}\setminus\{\mathbf{b}\})$. Then we continue by characterising states that have a just path in which an **a**-action is never

followed by a **b**-action. That is, we show that the formula that characterises the set of states that admit an **a**-**b**-liveness violation are exactly those states satisfying formula violate of Table 3.

Before we prove our claim that invariant exactly characterises states admitting just, **b**-free paths we first prove an auxiliary lemma. This auxiliary lemma claims that $\mathsf{elim}(\lambda)$ captures exactly those states that have a **b**-free path that eliminates action λ, and leads to a set of states represented by Y.

Lemma 38 *For all environments ϑ, states $s \in St$, actions $\lambda \in \mathcal{A}$ and sets $\mathcal{F} \subseteq St$, we have $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]}$ iff a state satisfying $\mathcal{F}$ can be reached from s via a finite* **b***-free path ending with an action that eliminates λ.*

Proof The construct is a standard construct in the modal μ-calculus; we refer to textbook proofs for the stated correspondence. □

We continue by substantiating the claim that invariant characterises the states admiting just, **b**-free paths. For the sake of conciseness, let $\mathcal{J}$ be a shorthand for $\mathcal{J}(\mathcal{A}\backslash\{\mathbf{b}\})$. The following lemma states that invariant exactly characterises the set of states $\mathcal{J}$.

Lemma 39 *For all $s \in St$ we have $s \in \mathcal{J}$ iff $s \in [\![\mathsf{invariant}]\!]$.*

Proof Let ϑ be an arbitrary environment. We first show, by showing mutual set inclusion, that $\mathcal{J}$ is a fixed point of the transformer $T_{\mathsf{invariant}}$ defined below:

$$T_{\mathsf{invariant}}(\mathcal{F}) = \bigcap_{\lambda \in \mathcal{A}} \{s \in St \mid \lambda \in \mathsf{En}(s) \Rightarrow s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]}\}$$

- Pick an arbitrary state $s \in \mathcal{J}$. Let π be a path that witnesses $s \in \mathcal{J}$. Pick an arbitrary action $\lambda \in \mathcal{A}$ and assume that $\lambda \in \mathsf{En}(s)$. We must show that $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{J}]}$ holds. From the fact that π witnesses $s \in \mathcal{J}$, we obtain that there must be some transition t on π such that $\lambda \not\rightsquigarrow \ell(t)$, i.e., $\ell(t) \in \#\lambda$, and by Proposition 36, $target(t) \in \mathcal{J}$. Hence we can conclude the desired $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{J}]}$.
- Pick a state $s \in T_{\mathsf{invariant}}$. Suppose $\mathsf{En}(s) = \emptyset$. Then state s itself is a just path and hence $s \in \mathcal{J}$. Next, suppose $\mathsf{En}(s) \neq \emptyset$ and let $\lambda \in \mathsf{En}(s)$. Then also $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{J}]}$. By Lemma 38, there must be some **b**-free finite path $s = s_0\, t_0\, s_1\, t_1 \ldots t_j\, s_{j+1}$ such that transition t_j eliminates λ and $s_{j+1} \in \mathcal{J}$. By Proposition 35, then also the path witnessing $s_{j+1} \in \mathcal{J}$, prefixed with $s_0\, t_0\, s_1\, t_1 \ldots t_j$, is a just path witnessing $s \in \mathcal{J}$.

We conclude that, indeed, $\mathcal{J}$ is a fixed point of $T_{\mathsf{invariant}}$. We next show that $\mathcal{J}$ is the greatest fixed point of $T_{\mathsf{invariant}}$; that is, for any $\mathcal{F}$ satisfying $T_{\mathsf{invariant}}(\mathcal{F}) = \mathcal{F}$, we have $\mathcal{F} \subseteq \mathcal{J}$. Let $\mathcal{F}$ be a fixed point of $T_{\mathsf{invariant}}$, and choose $s \in \mathcal{F}$. Our aim is to show that $s \in \mathcal{J}$. First, observe that since $\mathcal{F}$ is a fixed point of $T_{\mathsf{invariant}}$ and $s \in \mathcal{F}$, we can conclude $s \in T_{\mathsf{invariant}}(\mathcal{F})$.

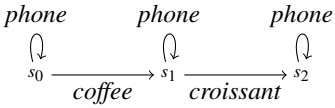
We construct a just, **b**-free path starting in state s by eliminating all actions enabled in s in an arbitrary but fixed order as follows. Let L denote the set of enabled actions in s. In case $L = \emptyset$, the state s itself witnesses $s \in \mathcal{J}$ and we are done. Otherwise, fix a total ordering $<$ on L. Pick the least action $\lambda \in L$. Since $s \in T_{\mathsf{invariant}}(\mathcal{F})$, also $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]}$ holds. Consequently, by Lemma 38, there is a finite **b**-free path $s_0\, t_0\, s_1\, t_1 \ldots t_j\, s_\lambda$ such that transition t_j eliminates λ and $s_\lambda \in \mathcal{F}$. Denote the set of enabled actions in s_λ by L_λ. Note that L_λ contains at least those actions of L that were not eliminated on some path from s to s_λ [it may, however, contain actions that were already eliminated on *some* path from s to s_λ, but, by Corollary 15, these actions were then not eliminated on *all* paths from s to s_λ

witnessing $s \in [\![\mathsf{elim}(\lambda)]\!]_{\vartheta[Y:=\mathcal{F}]}$.] We now repeat this construction by choosing the least $\lambda' \in \{\lambda'' \in L_\lambda \cap L \mid \lambda \prec \lambda''\}$, leading to a state $s_{\lambda'}$, *etcetera*, until we have constructed a finite path that eliminates all obligations in L and ends in a state $s' \in \mathcal{F}$. Note that this construction terminates since $|L| \leq |\mathcal{L}| < \infty$.

This means that for any state $s \in \mathcal{F}$, we can construct a finite path to another state in $\mathcal{F}$ such that all actions from $\mathsf{En}(s)$ are eliminated on that path. Since this holds invariantly for all states in $\mathcal{F}$, this construction can be repeated to yield a finite **b**-free just path or (in case it can be continued indefinitely) an infinite **b**-free just path starting in s. Hence, $s \in \mathcal{J}$ and therefore $\mathcal{F} \subseteq \mathcal{J}$. □

We illustrate the correspondence between invariant and $\mathcal{J}$ on the example we provided earlier.

Example 40 Reconsider Example 4, in which Alice drinks coffee and subsequently eats a croissant, Bob is engaged in a series of phone calls, and none of their activities interfere; see the following LTSC:



Suppose we claim that whenever Alice orders coffee, she eventually also orders a croissant. A counterexample to such a claim consists of a just path that contains a *coffee* event but is free of *croissant* actions following this *coffee* event. A state admits such a violating, *coffee*-less path iff it satisfies formula invariant.

We argue that in this case, s_1 does not satisfy formula invariant. To this end, we first show that s_1 does not satisfy $\mathsf{elim}(croissant)$. Notice that the set $\#croissant \backslash \{croissant\}$ is the empty set, while the set $\mathcal{A} \backslash (\#croissant \cup \{croissant\})$ is the set $\{coffee, phone\}$. Formula $\mathsf{elim}(croissant)$ therefore effectively holds in s_1 iff formula $\langle phone \rangle Q$ holds in state s_1. Due to the self-loop, this is the case exactly when state s_1 satisfies $\mathsf{elim}(croissant)$. Since this chain of reasoning must be continued indefinitely and we are looking for the least solution to Q, we must conclude that s_1 does not satisfy $\mathsf{elim}(croissant)$. As an immediate consequence we find that s_1 also does not satisfy invariant since *croissant* is one of the enabled actions in that state. Observe that this is in line with the fact that $s_1 \notin \mathcal{J}(\{coffee, phone\})$.

We now return to the original problem of characterising those states that have a just path that violates **a**-**b**-liveness. So far, we have established that formula invariant characterises those states that admit a **b**-free, just path. A state that admits a path violating **a**-**b**-liveness is therefore one that admits a finite path that, via an **a**-labelled transition, leads to a state satisfying invariant. Given the similarities with the formula for elim, we claim, without further proof, that formula violate indeed describes the set of states that admit an **a**-**b**-liveness violating just path.

Theorem 41 *Let* $(St, Tr, src, target, \ell, \rightsquigarrow)$ *be a finite LTSC with a concurrency-consistent labelling. Then all just paths starting in state* $s \in St$ *satisfy* **a**-**b**-*liveness if and only if* $s \notin [\![\mathsf{violate}]\!]$.

Example 42 We continue our previous example, showing that, indeed, the claim that whenever Alice orders coffee, she eventually also orders a croissant, holds true in state s_0.

We find that s_0 satisfies violate if, and only if, it satisfies $\langle coffee \rangle$invariant, $\langle coffee \rangle$violate, $\langle phone \rangle$violate, or $\langle croissant \rangle$violate. Notice that there is no *croissant* action enabled in s_0, so s_0 cannot satisfy $\langle croissant \rangle$violate. In order for s_0 to satisfy $\langle phone \rangle$violate, we require s_0 to again satisfy violate. Like before, such a cyclic chain of reasoning does not permit us to conclude that s_0 satisfies violate. Therefore, the only way to show that s_0 satisfies violate is to show that s_0 satisfies $\langle coffee \rangle$invariant. But as we may conclude from our previous example, also this will fail since s_1 does not satisfy invariant, which is required when we are to conclude that s_1 satisfies invariant. We can therefore conclude that state s_0 does not satisfy violate. Since the LTSC has a concurrency-consistent labelling, we may conclude by Theorem 41 that our liveness claim holds and Alice enjoys a croissant after drinking coffee.

Example 43 In Example 30, we concluded that all the conditions of Corollary 32 are satisfied for E_{Pet}, γ, *Pet* and the $\mathcal{C}(Pet)$-assignment (npc_ℓ, afc_ℓ), so the LTSC associated with *Pet* has a concurrency-consistent labelling. As a consequence, by Theorem 41 we can therefore conclude that the formula in Table 3, with $\mathcal{B} = \{\textbf{noncritA}, \textbf{noncritB}\}$, $\mathbf{a} = \textbf{noncritA}$ and $\mathbf{b} = \textbf{critA}$ expresses **noncritA**-**critA**-liveness.

9 Automated liveness analysis in mCRL2

A complete mCRL2 specification of Peterson's algorithm is listed in "Appendix C". The recursive specification E_{Pet} presented in Sect. 4 served as the starting point and the reader will easily recognise it under the mCRL2 keyword **proc**. That the mCRL2 specification looks somewhat more involved than the specification presented in Sect. 4 is because we have used some convenient extra features of mCRL2. Before we comment on these extra features, we emphasise that the use of these features is by no means essential. We could have also verified liveness for all just paths with the mCRL2 toolset with a specification that almost literally corresponds to the one presented in Sect. 4.

In an mCRL2 specification, labels can be parameterised with data, defined by means of an algebraic specification. In our specification we have included an enumerated type of which the elements correspond to the labels of Peterson's specification. This allows us to define, in a natural way, the functions npc_ℓ and afc_ℓ as mappings `npc` and `afc`, respectively, on the `Label` datatype. We then define a predicate `interfere(a,a')` that evaluates to true if, and only if, $npc_\ell(\texttt{a}) \cap afc_\ell(\texttt{a}') \neq \emptyset$ using the mappings `npc` and `afc`. In a similar vein, a predicate `blocking(a)` defines whether `a` is blocking or not.

The correspondence between labels and the data values representing them is achieved by turning the labels of *Pet* into *multi-actions*, 'labelling' the original actions with a parameterised action `label(<action>)`, where `<action>` identifies the original action. For instance, we represent the label **critA** using data value `a_critA`. In the equation defining `procA`, we have, instead of the occurrence of **critA** appearing in *procA* in Sect. 4, a multi-action `critA|label(a_critA)`. We can then choose to either hide the labels of the form `label(<label>)`, or hide the labels representing those in the specification of Peterson's algorithm in Sect. 4. The former allows us to generate a labelled transition system that is identical to that associated with *Pet*; the latter yields a labelled transition system in which transitions are labelled with actions of the form `label(<label>)`.

The toolset accepts the first-order modal μ-calculus of [12], which generalises the logic L_μ. With the labels available as a datatype and using the predicates `interfere` and `blocking`, we can express the formula expressing liveness for all just paths as an almost direct instantiation (with **noncritA** for **a** and **critA** for **b**) of the formula in

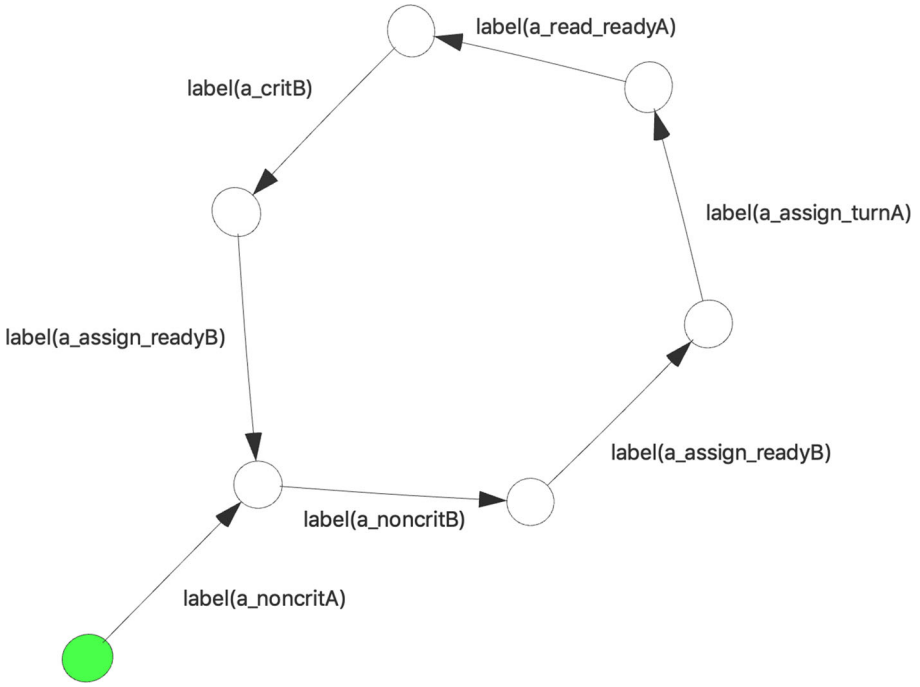


Fig. 2 Non-just liveness counterexample generated by the mCRL2 toolset

Table 3. The formula we have used to verify that the mCRL2 specification of Peterson's algorithm satisfies the required liveness property is listed in "Appendix D". The extra features of mCRL2 described above facilitate writing the generalised disjunctions and conjunctions as existential and universal quantifications. Note, however, that, since the quantifications are over finite sets, they can be replaced by finite disjunctions and conjunctions.

Verifying whether the mCRL2 specification of Peterson's algorithm satisfies **noncritA-critA**-liveness requires under half a second using the toolset and results in an affirmative verdict.[2] This once more confirms the manual correctness proof of [4]. If we modify the specification of the mapping `afc` by including `c_ReadyA` in `afc(a_read_readyA)`, `c_ReadyB` in `afc(a_read_readyB)`, and `c_Turn` in both `afc(a_read_turnA)` and `afc(a_read_turnB)`, then the toolset produces the counterexample shown in Fig. 2. Note that the modification corresponds to not treating these actions as signals and that the counterexample represents the non-just path discussed in Sect. 4.

[2] The mCRL2 sources can be found in the *academic* example directory of the mCRL2 repository, which can be obtained from https://github.com/mCRL2org/mCRL2, revision `b45856d9`.

10 Conclusions

To facilitate the automated verification of liveness properties, we have proposed a notion of concurrency-consistent labelling for labelled transition system with concurrency together with a formulation of justness in terms of states and actions. We have presented sufficient conditions on a process specification in a calculus with ACP-style communication that guarantee that the associated labelled transition system with concurrency has a concurrency-consistent labelling. Moreover, for LTSCs with a concurrency-consistent labelling we have shown how to formalise a liveness property under justness assumptions in the modal μ-calculus.

We have built on the firm foundation laid by van Glabbeek in [9], but had to slightly deviate from it to enable a special treatment of signal transitions in a regular process calculus. Furthermore, we essentially relied on the ACP-style communication mechanism in our calculus.

As an example of our theory, we have shown that Peterson's mutual exclusion algorithm can be specified in such a way that the associated LTSC has a concurrency-consistent labelling. Using the mCRL2 toolset we were able to verify that the specification satisfies the required liveness property for all just paths. We conjecture that similar specifications can be realised for the generalisation of Peterson's algorithm to N processes [17], and for Lamport's bakery algorithm [15]; it remains to confirm liveness properties for all just paths for these specifications with the mCRL2 toolset.

We see several directions in which our current work can be extended. For example, it would be useful to automate the verification of the syntactic conditions that guarantee that a specification induces an LTSC that has a concurrency-consistent labelling. A more challenging task is to identify to which extent the fragment of the process calculus can be extended without losing the guarantee that the LTSCs associated with expressions in that fragment have a concurrency-consistent labelling. We believe it may even be possible to check sufficient conditions for the LTSC to have a concurrency-consistent labelling by phrasing appropriate modal μ-calculus formulas. Finally, an open issue in the context of justness is the definition of behavioural equivalences, such as component-preserving variants of strong bisimilarity [16] or divergence-preserving branching bisimilarity [10]. The latter is a particularly interesting starting point because it deals with abstraction and is the coarsest congruence included in branching bisimilarity that distinguishes livelock from deadlock and is compatible with parallel composition [11].

Acknowledgements We thank the anonymous reviewers for their elaborate reviews and good suggestions. We thank Rob van Glabbeek for interesting discussions on the topic of this paper and for being a source of inspiration to us over the years.

Appendix A: Detailed proofs of lemmas in Sects. 3 and 5

In this "Appendix" we present elaborate proofs of Lemmas 6, 12 and 14, restated below as Lemmas 44, 45 and 46, respectively.

Lemma 44 *For all transitions t and v, if $src(t) = src(v)$ and $t \rightharpoonup v$, then there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $comp(u) = comp(t)$.*

Proof Let $P = src(t) = src(v)$ and suppose that $t \rightharpoonup v$, hence $comp(t) \cap comp(v) = \emptyset$. We prove with induction on v that there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $comp(u) = comp(t)$.

If the last rule applied in v is (PREF), (SUM-L), (SUM-R) or (REC), then $comp(v) = \{\epsilon\}$, and, due to the syntactic form of P, the last rule applied in t must also be one of these rules, so $comp(t) = \{\epsilon\}$. Thus, we find that $comp(t) \cap comp(v) = \{\epsilon\}$, contradicting the assumption of the lemma.

Suppose that the last rule applied in v is (PAR-L). Then there exist P_1 and P_2 such that $P = P_1 \parallel P_2$, and a subderivation v' of v such that $src(v') = P_1$, $\ell(v) = \ell(v')$, $target(v) = target(v') \parallel P_2$ and $comp(v) = \text{L} \rhd comp(v')$. From the syntactic shape of $src(t) = src(v) = P_1 \parallel P_2$ we conclude that the last rule applied in t must be (PAR-L), (PAR-R) or (COMM). We distinguish these three cases:

- If the last rule applied in t is (PAR-L), then t has a subderivation t' with $src(t') = P_1$ and $\ell(t') = \ell(t)$. Since $comp(t) = \text{L} \rhd comp(t')$ and $comp(v) = \text{L} \rhd comp(v')$, and $comp(t) \cap comp(v) = \emptyset$, we have that $comp(t') \cap comp(v') = \emptyset$, so $t' \rightharpoonup v'$. Hence, by the induction hypothesis, there exists a transition u' with $src(u') = target(v')$, $\ell(u') = \ell(t')$ and $comp(u') = comp(t')$. We can now construct from u' with an application of (PAR-L) a derivation u with $src(u) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $comp(u) = \text{L} \rhd comp(u') = \text{L} \rhd comp(t') = comp(t)$.
- If the last rule applied in t is (PAR-R), then t has a subderivation t' with $src(t') = P_2$ and $\ell(t') = \ell(t)$. Then with an application of (PAR-R) we can construct from t' a derivation u with $src(u) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \ell(t') = \ell(t)$ and $comp(u) = \text{R} \rhd comp(t') = comp(t)$.
- If the last rule applied in t is (COMM), then t has subderivations t_1 and t_2 with $src(t_1) = P_1$, $src(t_2) = P_2$, and $\gamma(\ell(t_1), \ell(t_2)) = \ell(t)$. From $comp(t) = \text{L} \rhd comp(t_1) \cup \text{R} \rhd comp(t_2)$ and $comp(t) \cap comp(v) = \emptyset$, we conclude that $comp(t_1) \cap comp(v') = \emptyset$, so $t_1 \rightharpoonup v'$, and hence, by the induction hypothesis, there exists a derivation u_1 with $src(u_1) = target(v')$, $\ell(u_1) = \ell(t_1)$ and $comp(u_1) = comp(t_1)$. From u_1 and t_2 we can now, with an application of (COMM), construct a derivation u with $src(u) = src(u_1) \parallel src(t_2) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \gamma(\ell(u_1), \ell(t_2)) = \gamma(\ell(t_1), \ell(t_2)) = \ell(t)$ and $comp(u) = \text{L} \rhd comp(u_1) \cup \text{R} \rhd comp(t_2) = \text{L} \rhd comp(t_1) \cup \text{R} \rhd comp(t_2) = comp(t)$.

If the last rule applied in v is (PAR-R), then the argument is symmetric to the argument for the case that the last rule applied in v is (PAR-L).

Suppose last rule applied in v is (COMM). Then there exist subderivations v_1 and v_2 of v with $src(v) = src(v_1) \parallel src(v_2)$, $\ell(v) = \gamma(\ell(v_1), \ell(v_2)$, $target(v) = target(v_1) \parallel target(v_2)$ and $comp(v) = \text{L} \rhd comp(v_1) \cup \text{R} \rhd comp(v_2)$. From the syntactic shape of $src(t) = src(v) = src(v_1) \parallel src(v_2)$, we conclude that the last rule applied in t must be (PAR-L), (PAR-R) or (COMM). We distinguish these three cases:

- If the last rule applied in t is (PAR-L), then t has a subderivation t' with $src(t') = src(v_1)$ and $\ell(t') = \ell(t)$. Since $comp(t) = \text{L} \rhd comp(t')$, $comp(v) = \text{L} \rhd comp(v_1) \cup \text{R} \rhd comp(v_2)$,

and $comp(t) \cap comp(v) = \emptyset$, we have that $comp(t') \cap comp(v_1) = \emptyset$. So $t' \frown\!\bullet\, v_1$, and hence, by the induction hypothesis, there exists a transition u' with $src(u') = target(v_1)$, $\ell(u') = \ell(t')$ and $comp(u') = comp(t')$. We can now construct from u' with an application of (PAR-L) a derivation u with $src(u) = src(u') \parallel target(v_2) = target(v_1) \parallel target(v_2) = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $comp(u) = \mathrm{L} \rhd comp(u') = \mathrm{L} \rhd comp(t') = comp(t)$.

- If the last rule applied in t is (PAR-R), then the argument is similar to the argument in the previous case, using the induction hypothesis for v_2 instead.
- If the last rule applied in t is (COMM), then t has subderivations t_1 and t_2 with $src(t_1) = src(v_1)$, $src(t_2) = src(v_2)$, $\gamma(\ell(t_1), \ell(t_2)) = \ell(t)$. Since $comp(t) = \mathrm{L} \rhd comp(t_1) \cup \mathrm{R} \rhd comp(t_2)$ and $comp(v) = \mathrm{L} \rhd comp(v_1) \cup \mathrm{R} \rhd comp(v_2)$, and $comp(t) \cap comp(v) = \emptyset$, we have that $comp(t_1) \cap comp(v_1) = \emptyset$ and $comp(t_2) \cap comp(v_2) = \emptyset$. Hence, by the induction hypothesis, there exist action transitions u_1 and u_2 with $src(u_1) = target(v_1)$, $src(u_2) = target(v_2)$, $\ell(u_1) = \ell(t_1)$, $\ell(u_2) = \ell(t_2)$, $comp(u_1) = comp(t_1)$ and $comp(u_2) = comp(t_2)$. We can now construct from u_1 and u_2 with an application of (COMM) a derivation u with $src(u) = src(u_1) \parallel src(u_2) = target(v_1) \parallel target(v_2) = target(v)$, $\ell(u) = \gamma(\ell(u_1), \ell(u_2)) = \gamma(\ell(t_1), \ell(t_2)) = \ell(t)$, and $comp(u) = \mathrm{L} \rhd comp(u_1) \cup \mathrm{R} \rhd comp(u_2) = \mathrm{L} \rhd comp(t_1) \cup \mathrm{R} \rhd comp(t_2) = comp(t)$.

Suppose that the last rule applied in v is (ENC). Then there exists a subderivation v' with $src(v) = \partial_H(src(v'))$ for some $H \subseteq \mathcal{L}$, $\ell(v') = \ell(v) \notin H$ and $comp(v) = comp(v')$. From the syntactic shape of $src(t) = src(v) = \partial_H(src(v'))$ it follows that the last rule applied in t must be (ENC) too. So t has a subderivation t' with $src(t') = src(v')$ and $\ell(t') = \ell(t)$. Since $comp(t) = comp(t')$ and $comp(v) = comp(v')$, from $comp(t) \cap comp(v) = \emptyset$ it follows that $comp(t') \cap comp(v') = \emptyset$. Hence, by the induction hypothesis, there exists u' with $src(u') = target(v')$, $\ell(u') = \ell(t')$ and $comp(u') = comp(t')$. With an application of (ENC) we can now construct from u' a derivation u with $src(u) = \partial_H(src(u')) = \partial_H(target(v')) = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $comp(u) = comp(u') = comp(t') = comp(t)$. □

Lemma 45 *If the communication function γ is signal-respecting, then a transition t is a signal transition if, and only if, $afc(t) = \emptyset$.*

Proof We prove with induction on the derivation t that t is a signal transition if, and only if, $afc(t) = \emptyset$.

If the last rule applied in t is (PREF), then $src(t) \neq target(t)$ so t is not a signal transition and $afc(t) \neq \emptyset$.

If the last rule applied in t is (SUM-L), (SUM-R) or (REC), then t is a signal transition if, and only if, $\ell(t) \in \mathcal{S}$ and $src(t) = target(t)$, if, and only if, $afc(t) = \emptyset$.

If the last rule applied in t is (PAR-L), then t has a subderivation t' such that, for some process expression P, $src(t) = src(t') \parallel P$, $target(t) = target(t') \parallel P$ and $\ell(t) = \ell(t')$. On the one hand, if t is a signal transition, then from $src(t) = target(t)$ it follows that $src(t') = target(t')$ and $\ell(t') = \ell(t) \in \mathcal{S}$, so t' is a signal transition too. By the induction hypothesis, $afc(t') = \emptyset$, and, since $afc(t) = \mathrm{L} \rhd afc(t')$, it follows that $afc(t) = \emptyset$. On the other hand, if $afc(t) = \emptyset$, then $afc(t') = \emptyset$. So by the induction hypothesis it follows that t' is a signal transition. So $src(t') = target(t')$ and hence $src(t) = target(t)$, and therefore t is a signal transition too.

If the last rule applied in t is (PAR-R), then the argument is similar to the argument in the previous case.

If the last rule applied in t is (COMM), then there exist subderivations t_1 and t_2 such that $src(t) = src(t_1) \parallel src(t_2)$, $target(t) = target(t_1) \parallel target(t_2)$, and $\gamma(\ell(t_1), \ell(t_2)) = \ell(t)$.

On the one hand, if t is a signal transition, then from $src(t) = target(t)$ it follows that $src(t_1) = target(t_1)$ and $src(t_2) = target(t_2)$ and, since γ is signal-respecting, from $\ell(t) \in \mathcal{S}$ it follows that $\ell(t_1), \ell(t_2) \in \mathcal{S}$. Hence both t_1 and t_2 are signal transitions too. By the induction hypothesis, $afc(t_1) = afc(t_2) = \emptyset$, and since $afc(t) = \text{L} \rhd afc(t_1) \cup \text{R} \rhd afc(t_2)$, it follows that $afc(t) = \emptyset$. On the other hand, if $afc(t) = \emptyset$, then since $afc(t) = \text{L} \rhd afc(t_1) \cup \text{R} \rhd afc(t_2)$, it follows that $afc(t_1) = afc(t_2) = \emptyset$, so, by the induction hypothesis, t_1 and t_2 are signal transitions. Hence, $src(t_1) = target(t_1)$, $src(t_2) = target(t_2)$ and $\ell(t_1), \ell(t_2) \in \mathcal{S}$. It follows that $src(t) = src(t_1) \parallel src(t_2) = target(t_1) \parallel target(t_2) = target(t)$ and, since γ is signal-respecting, $\ell(t) = \gamma(\ell(t_1), \ell(t_2)) \in \mathcal{S}$, so t is a signal transition.

If the last rule applied in t is (ENC), then there exists a subderivation t' such that $src(t) = \partial_H(src(t'))$ for some $H \subseteq \mathcal{L}$, $\ell(t') = \ell(t)$ and $target(t) = \partial_H(src(t'))$. Furthermore, note that $afc(t') = afc(t)$. On the one hand, if t is a signal transition, then t' is a signal transition too, so by the induction hypothesis, $afc(t') = \emptyset$. Since $afc(t) = afc(t')$, it follows that $afc(t) = \emptyset$. On the other hand, if $afc(t) = \emptyset$, then $afc(t') = \emptyset$. So, by the induction hypothesis, t' is a signal transition, and hence $src(t') = target(t')$ and $\ell(t') \in \mathcal{S}$. It follows that $src(t) = target(t)$ and $\ell(t) \in \mathcal{S}$, so t is a signal transition. □

Lemma 46 *For all transitions t and v, if $src(t) = src(v)$ and $npc(t) \cap afc(v) = \emptyset$, then there exists a transition u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $npc(u) = npc(t)$. If γ is signal-respecting and t is an action transition, then so is u.*

Proof Let $P = src(t) = src(v)$ and suppose that $npc(t) \cap afc(v) = \emptyset$. We prove with induction on v that there exists u with $src(u) = target(v)$, $\ell(u) = \ell(t)$ and $npc(u) = npc(t)$.

If the last rule applied in v is (PREF), then there exist λ and P' such that $P = \lambda.P'$ and $afc(v) = \{\epsilon\}$. Due to the syntactic form of P, the last rule applied in t must also be (PREF) and therefore $npc(t) = \{\epsilon\}$. Thus, we find that $npc(t) \cap afc(v) = \{\epsilon\}$, contradicting the assumption of the lemma.

If the last rule applied in v is (SUM-L) or (SUM-R). Then either then $P = P_1 + P_2$, so the last rule applied in t is also (SUM-L) or (SUM-R). Since $npc(t) = \{\epsilon\}$ and $npc(t) \cap afc(v) = \emptyset$, we have that $afc(v) \neq \{\epsilon\}$, and hence $afc(v) = \emptyset$. So, by Lemma 12, v is a signal transition, and hence $\lambda \in \mathcal{S}$ and $P = P'$. Then we have that $src(t) = src(v) = target(v)$, so we can take $u = t$ to satisfy the requirements of the lemma. Clearly, if t is an action transition, then so is u.

If the last rule applied in v is (REC), then $P = A$, so also the last rule applied in t is (REC). Since $npc(t) = \{\epsilon\}$ and $npc(t) \cap afc(v) = \emptyset$, it we have that $afc(v) = \emptyset$. So, by Lemma 12, v is a signal transition, and hence $P' = A$ and $\lambda \in \mathcal{S}$. Then we have that $src(t) = src(v) = target(v)$, so we can take $u = t$ to satisfy the requirements of the lemma. Clearly, if t is an action transition, then u is an action transition too.

Suppose that the last rule applied in v is (PAR-L). Then there exists a subderivation v' of v and a process expression P' such that $src(v) = src(v') \parallel P'$, $\ell(v) = \ell(v')$ and $afc(v) = \text{L} \rhd afc(v')$. From the syntactic shape of $src(t) = src(v) = src(v') \parallel P'$ we conclude that the last rule applied in t must be (PAR-L), (PAR-R) or (COMM). We distinguish these three cases:

- If the last rule applied in t is (PAR-L), then t has a subderivation t' with $src(t') = P_1$ and $\ell(t') = \ell(t)$. Since $npc(t) = \text{L} \rhd npc(t')$ and $afc(v) = \text{L} \rhd afc(v')$, it follows from $npc(t) \cap afc(v) = \emptyset$ that $npc(t') \cap afc(v') = \emptyset$. Hence, by the induction hypothesis, there exists a transition u' with $src(u') = target(v')$, $\ell(u') = \ell(t')$ and $npc(u') = npc(t')$. We can now construct from u' with an application of (PAR-L) a derivation u with $src(u) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $npc(u) = \text{L} \rhd npc(u') = \text{L} \rhd npc(t') = npc(t)$.

It remains to argue that if γ is signal-respecting and t is an action transition, then u is an action transition too. To this end, first note that if γ signal-respecting and t is an action transition, then, by Lemma 12, $afc(t) \neq \emptyset$, say $\mathrm{L}\sigma \in afc(t)$ for some $\sigma \in \mathcal{C}^*$. Then, since $afc(t) = \mathrm{L} \rhd afc(t')$, it follows that $\sigma \in afc(t')$, so t' is an action transition too. Then, by the induction hypothesis, also u' is an action transition, so there exists $\sigma \in afc(u')$, and hence $\mathrm{L}\sigma' \in afc(u)$, which therefore is also an action transition.

- If the last rule applied in t is (PAR-R), then t has a subderivation t' with $src(t') = P_2$ and $\ell(t') = \ell(t)$. Then with an application of (PAR-R) we can construct from t' a derivation u with $src(u) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \ell(t') = \ell(t)$ and $npc(u) = \mathrm{R} \rhd npc(t') = npc(t)$.
  If t is an action transition, then, by Lemma 12, $afc(t) \neq \emptyset$, and hence there exists $\sigma \in \mathcal{C}^*$ such that $\mathrm{R}\sigma \in afc(t)$. From the construction of u it is then easy to see that also $\mathrm{R}\sigma \in afc(u)$, proving that u is an action transition too.
- If the last rule applied in t is (COMM), then t has subderivations t_1 and t_2 with $src(t_1) = P_1$, $src(t_2) = P_2$, and $\gamma(\ell(t_1), \ell(t_2)) = \ell(t)$. From $npc(t) = \mathrm{L} \rhd npc(t_1) \cup \mathrm{R} \rhd npc(t_2)$ and $npc(t) \cap afc(v) = \emptyset$ we conclude that $npc(t_1) \cap afc(v') = \emptyset$, so by the induction hypothesis there exists a derivation u_1 with $src(u_1) = src(v')$, $\ell(u_1) = \ell(t_1)$ and $npc(u_1) = npc(t_1)$. From u_1 and t_2 we can now, with an application of (COMM), construct a derivation u with $src(u) = src(u_1) \parallel src(t_2) = target(v') \parallel P_2 = target(v)$, $\ell(u) = \gamma(\ell(u_1), \ell(t_2)) = \gamma(\ell(t_1), \ell(t_2)) = \ell(t)$ and $npc(u) = \mathrm{L} \rhd npc(u_1) \cup \mathrm{R} \rhd npc(t_2) = \mathrm{L} \rhd npc(t_1) \cup \mathrm{R} \rhd npc(t_2) = npc(t)$.
  If γ is signal-respecting and t is an action transition, then, by Lemma 12, $afc(t) \neq \emptyset$. Hence, since $afc(t) = \mathrm{L} \rhd afc(t_1) \cup \mathrm{R} \rhd afc(t_2)$, we have that $afc(t_1) \neq \emptyset$ or $afc(t_2) \neq \emptyset$. In the first case, t_1 is an action transition, so we get from the induction hypothesis that u_1 is an action transition. It follows that $afc(u_1) \neq \emptyset$, and hence $afc(u) \neq \emptyset$, so u is an action transition. In the second case, we simply get that $\mathrm{R} \rhd afc(t_2) \subseteq afc(u)$, so $afc(u) \neq \emptyset$ and therefore u is an action transition.

If the last rule applied in v is (PAR-R), then the argument is symmetric to the argument for the case that the last rule applied in v is (PAR-L).

Suppose that the last rule applied in v is (COMM). Then there exist subderivations v_1 and v_2 of v with $src(v) = src(v_1) \parallel src(v_2)$, $\ell(v) = \gamma(\ell(v_1), \ell(v_2)$ and $afc(v) = \mathrm{L} \rhd afc(v_1) \cup \mathrm{R} \rhd afc(v_2)$. From the syntactic shape of $src(t) = src(v) = src(v_1) \parallel src(v_2)$, we conclude that the last rule applied in t must be (PAR-L), (PAR-R) or (COMM). We distinguish these three cases:

- If the last rule applied in t is (PAR-L), then t has a subderivation t' with $src(t') = src(v_1)$ and $\ell(t') = \ell(t)$. Since $npc(t) = \mathrm{L} \rhd npc(t')$ and $afc(v) = \mathrm{L} \rhd afc(v_1) \cup \mathrm{R} \rhd afc(v_2)$, it follows from $npc(t) \cap afc(v) = \emptyset$ that $npc(t') \cap afc(v_1) = \emptyset$. So, by the induction hypothesis, there exists a transition u' with $src(u') = target(v_1)$, $\ell(u') = \ell(t')$ and $npc(u') = npc(t')$. We can now construct from u' with an application of (PAR-L) a derivation u with $src(u) = src(u') \parallel target(v_2) = target(v_1) \parallel target(v_2) = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $npc(u) = \mathrm{L} \rhd npc(u') = \mathrm{L} \rhd npc(t') = npc(t)$.
  If γ is signal-respecting and t is an action transition, then, by Lemma 12, we have that $afc(t) \neq \emptyset$, and hence $afc(t') \neq \emptyset$. So t' is an action transition, and hence, by the induction hypothesis, u' is an action transition. Therefore, by construction, u is an action transition too.
- If the last rule applied in t is (PAR-R), then the argument is similar to the argument in the previous case, using the induction hypothesis for v_2 instead.

- If the last rule applied in t is (COMM), then t has subderivations t_1 and t_2 with $src(t_1) = src(v_1)$, $src(t_2) = src(v_2)$, $\gamma(\ell(t_1), \ell(t_2)) = \ell(t)$. Since $npc(t) = L \rhd npc(t_1) \cup \mathrm{R} \rhd npc(t_2)$ and $afc(v) = L \rhd afc(v_1) \cup \mathrm{R} \rhd afc(v_2)$, it follows from $npc(t) \cap afc(v) = \emptyset$ that $npc(t_1) \cap afc(v_1) = \emptyset$ and $npc(t_2) \cap afc(v_2) = \emptyset$. Hence, by the induction hypothesis, there exist action transitions u_1 and u_2 with $src(u_1) = target(v_1)$, $src(u_2) = target(v_2)$, $\ell(u_1) = \ell(t_1)$, $\ell(u_2) = \ell(t_2)$, $npc(u_1) = npc(t_1)$ and $npc(u_2) = npc(v_2)$. We can now construct from u_1 and u_2 with an application of (COMM) a derivation u with $src(u) = src(u_1) \parallel src(u_2) = target(v_1) \parallel target(v_2) = target(v)$, $\ell(u) = \gamma(\ell(u_1), \ell(u_2)) = \gamma(\ell(t_1), \ell(t_2)) = \ell(t)$, and $npc(u) = \mathrm{L} \rhd npc(u_1) \cup \mathrm{R} \rhd npc(u_2) = \mathrm{L} \rhd npc(t_1) \cup \mathrm{R} \rhd npc(t_2) = npc(t)$.
  If γ is signal-respecting and t is an action transition, then, by Lemma 12, $afc(t) \neq \emptyset$. Hence, since $afc(t) = \mathrm{L} \rhd afc(t_1) \cup \mathrm{R} \rhd afc(t_2)$, we have that $afc(t_1) \neq \emptyset$ or $afc(t_2) \neq \emptyset$. In the first case, t_1 is an action transition, so we get from the induction hypothesis that u_1 is an action transition. It follows that $afc(u_1) \neq \emptyset$, and hence $afc(u) \neq \emptyset$, so u is an action transition. In the second case, t_2 is an action transition, so we get from the induction hypothesis that u_2 is an action transition. It follows that $afc(u_2) \neq \emptyset$, and hence $afc(u) \neq \emptyset$, so u is an action transition.

Suppose that the last rule applied in v is (ENC). Then there exists a subderivation v' with $src(v) = \partial_H(src(v'))$ for some $H \subseteq \mathcal{L}$, $\ell(v') = \ell(v) \notin H$ and $npc(v) = npc(v')$. From the syntactic shape of $src(t) = src(v) = \partial_H(src(v'))$ it follows that the last rule applied in t must be (ENC) too. So t has a subderivation t' with $src(t') = src(v')$ and $\ell(t') = \ell(t)$. Since $npc(t) = npc(t')$ and $afc(v) = afc(v')$, from $npc(t) \cap afc(v) = \emptyset$ it follows that $npc(t') \cap afc(v') = \emptyset$. Hence, by the induction hypothesis, there exists u' with $src(u') = target(v')$, $\ell(u') = \ell(t')$ and $npc(u') = npc(t')$. With an application of (ENC) we can now construct from u' a derivation u with $src(u) = \partial_H(src(u')) = \partial_H(target(v')) = target(v)$, $\ell(u) = \ell(u') = \ell(t') = \ell(t)$ and $npc(u) = npc(u') = npc(t') = npc(t)$.

If γ is signal-respecting and t is an action transition, then, by Lemma 12 and $afc(t') = afc(t)$, t' is an action transition too, so by the induction hypothesis also u' is an action transition, and thus it follows, by Lemma 12 and $afc(u) = afc(u')$, that u is an action transition. □

B Detailed proof of a lemma in Sect. 7

In this "Appendix", we present a detailed proof of Lemma 27, restated below as Lemma 47.

Lemma 47 *Let E be a sequential recursive specification and let P be a parallel-sequential process expression over E. If P' is reachable from P, then $\mathcal{C}(P') = \mathcal{C}(P)$ and $P'|_\sigma$ is reachable from $P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.*

Proof Let us first consider the special case that there is a transition t such that $src(t) = P$ and $target(t) = P'$. With induction on t we establish that $\mathcal{C}(P') = \mathcal{C}(P)$ and $P\restriction_\sigma = P|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.

If the last rule applied in t is (PREF), (SUM- L), (SUM- R) or (REC), then P is a sequential process expression and, since E is a sequential recursive specification, so is P'. It follows that $\mathcal{C}(P') = \{\epsilon\} = \mathcal{C}(P)$ and $P\restriction_\epsilon = P'$ is reachable from $P = P|_\epsilon$.

If the last rule applied in t is (PAR-L), then there exist P_1, P_1' and P_2 such that $P = P_1 \parallel P_2$, $P' = P_1' \parallel P_2$, and t has a subderivation t' with $src(t') = P_1$ and $target(t') = P_1'$. By the induction hypothesis, $\mathcal{C}(P_1) = \mathcal{C}(P_1')$ and $P_1|_\sigma = P_1'|_\sigma$ for all $\sigma \in \mathcal{C}(P_1)$. It follows that

$\mathcal{C}(P) = L \rhd \mathcal{C}(P_1) \cup_{\mathrm{R}} \rhd \mathcal{C}(P_2) = L \rhd \mathcal{C}(P_1') \cup_{\mathrm{R}} \rhd \mathcal{C}(P_2) = \mathcal{C}(P')$. Moreover, since $P_1|_\sigma = P_1'|_\sigma$ for all $\sigma \in \mathcal{C}(P_1)$, it also follows that $P|_\sigma = P'|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.

If the last rule applied in t is (PAR-R), then the argument is analogous to the argument in the case that the last rule applied is (PAR-L).

If the last rule applied in t is (COMM), then there exist P_1, P_1', P_2 and P_2' such that $P = P_1 \parallel P_2$, $P' = P_1' \parallel P_2'$, and t has a subderivations t_1 and t_2 with $src(t_1) = P_1$, $target(t_1) = P_1'$, $src(t_2) = P_2$ and $target(t_2) = P_2'$. By the induction hypothesis, $\mathcal{C}(P_1) = \mathcal{C}(P_1')$, $\mathcal{C}(P_2) = \mathcal{C}(P_2')$, $P_1|_\sigma = P_1'|_\sigma$ for all $\sigma \in \mathcal{C}(P_1)$, and $P_2|_\sigma = P_2'|_\sigma$ for all $\sigma \in \mathcal{C}(P_2)$. It follows that $\mathcal{C}(P) = L \rhd \mathcal{C}(P_1) \cup_{\mathrm{R}} \rhd \mathcal{C}(P_2) = L \rhd \mathcal{C}(P_1') \cup_{\mathrm{R}} \rhd \mathcal{C}(P_2') = \mathcal{C}(P')$. Moreover, since $P_1|_\sigma = P_1'|_\sigma$ for all $\sigma \in \mathcal{C}(P_1)$ and $P_2|_\sigma = P_2'|_\sigma$ for all $\sigma \in \mathcal{C}(P_2)$, it also follows that $P|_\sigma = P'|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.

If the last rule applied in t is (ENC), then there exist P_1 and P_1' such that $P = \partial_H(P_1)$ and $P' = \partial_H(P_1')$, and t has a subderivation t' with $src(t') = P_1$ and $target(t') = P_1'$. By the induction hypothesis, $\mathcal{C}(P) = \mathcal{C}(P_1) = \mathcal{C}(P_1') = \mathcal{C}(P')$ and $P|_\sigma = P_1|_\sigma = P_1'|_\sigma = P'|_\sigma$ for all $\sigma \in \mathcal{C}(P)$.

Now, if P' is reachable from P, then the statement of the lemma follows with a straightforward induction on the number of transitions in a path from P to P'. □

C mCRL2 specification of Peterson's algorithm

```
sort Label =
  struct
    a_assign_readyA | a_read_readyA | a_assign_readyB| a_read_readyB
 |
    a_assign_turnA | a_assign_turnB | a_read_turnA | a_read_turnB |
    a_critA | a_critB | a_noncritA | a_noncritB
  ;

sort
  TurnType= struct A | B;
act
  assign_readyA,r_assign_readyA,s_assign_readyA: Bool;
  assign_readyB,r_assign_readyB,s_assign_readyB: Bool;
  assign_turnA,r_assign_turnA,s_assign_turnA;
  assign_turnB,r_assign_turnB,s_assign_turnB;
  read_readyA,r_read_readyA,s_read_readyA: Bool;
  read_readyB,r_read_readyB,s_read_readyB: Bool;
  read_turnA,r_read_turnA,s_read_turnA;
  read_turnB,r_read_turnB,s_read_turnB;
  noncritA, critA, noncritB, critB;
  label:Label;

proc
  ReadyA(b: Bool) =
    sum b':Bool.r_assign_readyA(b').ReadyA(b')
  + s_read_readyA(b)|label(a_read_readyA).ReadyA();

  ReadyB(b: Bool) =
    sum b':Bool.r_assign_readyB(b').ReadyB(b')
  + s_read_readyB(b)|label(a_read_readyB).ReadyB();

  Turn(t: TurnType) =
    r_assign_turnA.Turn(A)
  + r_assign_turnB.Turn(B)
  + (t==A) -> s_read_turnA|label(a_read_turnA).Turn()
  + (t==B) -> s_read_turnB|label(a_read_turnB).Turn();

  procA =
    noncritA|label(a_noncritA).
```

```
      s_assign_readyA(true)|label(a_assign_readyA).
        s_assign_turnB|label(a_assign_turnB).
      sum rA,rB: Bool, t:TurnType.
        (r_read_readyB(false)+r_read_turnA).
              critA|label(a_critA).
            s_assign_readyA(false)|label(a_assign_readyA).procA;

  procB =
    noncritB|label(a_noncritB).
      s_assign_readyB(true)|label(a_assign_readyB).
        s_assign_turnA|label(a_assign_turnA).
      sum rA,rB: Bool, t:TurnType.
        (r_read_readyA(false)+r_read_turnB).
          critB|label(a_critB).
            s_assign_readyB(false)|label(a_assign_readyB).procB;

init
hide({assign_readyA,read_readyA,assign_readyB,read_readyB,
      assign_turnA,assign_turnB,read_turnA,read_turnB,
      critA,critB,noncritA,noncritB},
  allow({assign_readyA|label, read_readyA|label,
     assign_readyB|label, read_readyB|label,
     assign_turnA|label, assign_turnB|label,
     read_turnA|label, read_turnB|label,
     critA|label, critB|label, noncritA|label, noncritB|label},
      comm({r_assign_readyA|s_assign_readyA->assign_readyA,
        r_read_readyA|s_read_readyA -> read_readyA,
            r_assign_readyB|s_assign_readyB->assign_readyB,
        r_read_readyB|s_read_readyB -> read_readyB,
        r_assign_turnA|s_assign_turnA ->assign_turnA,
        r_assign_turnB|s_assign_turnB ->assign_turnB,
        r_read_turnA|s_read_turnA -> read_turnA,
        r_read_turnB|s_read_turnB -> read_turnB},
          procA||procB||ReadyA(false)||ReadyB(false)||Turn(A)
      )
  )
);

sort Labels=Set(Label);

sort Component =
  struct
    c_ReadyA | c_ReadyB | c_Turn | c_procA | c_procB
  ;
sort Components=Set(Component);
map
  npc:Label -> Components;
eqn
  npc(a_assign_readyA)={c_procA,c_ReadyA};
  npc(a_read_readyA)={c_procB, c_ReadyA};
  npc(a_assign_readyB)={c_procB, c_ReadyB};
  npc(a_read_readyB)={c_procA, c_ReadyB};
  npc(a_assign_turnA)={c_procB, c_Turn};
  npc(a_assign_turnB)={c_procA, c_Turn};
  npc(a_read_turnA)={c_procA, c_Turn};
  npc(a_read_turnB)={c_procB, c_Turn};
  npc(a_critA)={c_procA};
  npc(a_critB)={c_procB};
  npc(a_noncritA)={c_procA};
  npc(a_noncritB)={c_procB};

map
  afc:Label -> Components;
eqn
  afc(a_assign_readyA)={c_procA, c_ReadyA};
  afc(a_read_readyA)={c_procB};
%  afc(a_read_readyA)={c_procB, c_ReadyA};
  afc(a_assign_readyB)={c_procB, c_ReadyB};
  afc(a_read_readyB)={c_procA};
%  afc(a_read_readyB)={c_procA, c_ReadyB};
```

```
  afc(a_assign_turnA)={c_procB, c_Turn};
  afc(a_assign_turnB)={c_procA, c_Turn};
  afc(a_read_turnA)={c_procA};
%  afc(a_read_turnA)={c_procA, c_Turn};
  afc(a_read_turnB)={c_procB};
%  afc(a_read_turnB)={c_procB, c_Turn};
  afc(a_critA)={c_procA};
  afc(a_critB)={c_procB};
  afc(a_noncritA)={c_procA};
  afc(a_noncritB)={c_procB};

map
  elim: Label -> Labels;
  interfere: Label#Label -> Bool;
var
  a,a': Label;
eqn
  elim(a) = {a'':Label | exists c:Component. c in npc(a'') && c in
 afc(a)};
  interfere(a,a')=exists c:Component.(c in npc(a) && c in afc(a'));

map
  blocking:Label -> Bool;
eqn
  blocking(a_noncritA)=true;
  blocking(a_noncritB)=true;
  blocking(a_assign_readyA)=false;
  blocking(a_read_readyA)=false;
  blocking(a_assign_readyB)=false;
  blocking(a_read_readyB)=false;
  blocking(a_assign_turnA)=false;
  blocking(a_assign_turnB)=false;
  blocking(a_read_turnA)=false;
  blocking(a_read_turnB)=false;
  blocking(a_critA)=false;
  blocking(a_critB)=false;
```

D Formula expressing liveness for all just paths

```
! mu W. (
    <action(a_noncritA)>(
      nu Y. forall a:Action.(val(!blocking(a)) && <action(a)>true) =>
        mu Q. (
      ( exists a':Action.val(interfere(a,a') && (a'!=a_critA)) &&
<action(a')>Y)
        ||
      ( exists a':Action.val(!interfere(a,a') && (a'!=a_critA)) &&
<action(a')>Q )
    )
    )
  ||
    exists a:Action.<action(a)>W
  )
```

References

1. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theor. Comput. Sci. **37**, 77–121 (1985)
2. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems—improvements in expressivity and usability. In: TACAS (2), volume 11428 of Lecture Notes in Computer Science, pp. 21–39. Springer (2019)
3. Cranen, S., Luttik, B., Willemse, T.A.C.: Evidence for fixpoint logic. In: CSL, volume 41 of LIPIcs, pp. 8–93. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
4. Dyseryn, V., van Glabbeek, R.J., Höfner, P.: Analysing mutual exclusion using process algebra with signals. In: Peters, K., Tini, S. (eds.) Proceedings Combined 24th International Workshop on Expressiveness

in Concurrency and 14th Workshop on Structural Operational Semantics and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017, volume 255 of EPTCS, pp. 18–34 (2017)
5. Emerson, E.A., Lei, C.-L.: Modalities for model checking: branching time logic strikes back. Sci. Comput. Program. **8**(3), 275–306 (1987)
6. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT **15**(2), 89–107 (2013)
7. van Glabbeek, R.J., Höfner, P.: CCS: it's not fair!—fair schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions. Acta Inf. **52**(2–3), 175–205 (2015)
8. van Glabbeek, R.J., Höfner, P.: Progress, justness, and fairness. ACM Comput. Surv. **52**(4), 69:1–69:38 (2019)
9. van Glabbeek, R.J.: Justness—a completeness criterion for capturing liveness properties (extended abstract). In: FoSSaCS, volume 11425 of Lecture Notes in Computer Science, pp. 505–522. Springer (2019)
10. van Glabbeek, R.J., Luttik, B., Trcka, N.: Branching bisimilarity with explicit divergence. Fundam. Inform. **93**(4), 371–392 (2009)
11. van Glabbeek, R.J., Luttik, B., Trcka, N.: Computation tree logic with deadlock detection. Log. Methods Comput. Sci. **5**(4) (2009)
12. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Sci. Comput. Program. **56**(3), 251–273 (2005)
13. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
14. Kozen, D.: Results on the propositional μ-calculus. Theoret. Comput. Sci. **27**(3), 333–354 (1982)
15. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
16. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) Theoretical Computer Science. Lecture Notes in Computer Science, vol. 104, pp. 167–183. Springer, Berlin (1981)
17. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
18. Wesselink, W., Willemse, T.A.C.: Evidence extraction from parameterised Boolean equation systems. In: ARQNL@IJCAR, volume 2095 of CEUR Workshop Proceedings, pp. 86–100. CEUR-WS.org (2018)