Off-the-Shelf Automated Analysis of Liveness Properties for Just Paths

Mark Bouwman, Bas Luttik, Tim A.C. Willemse

Off-the-shelf Automated Analysis of Liveness Properties for Just Paths
(extended abstract)

Mark Bouwman, Bas Luttik, and Tim Willemse
{m.s.bouwman,s.p.luttik,t.a.c.willemse}@tue.nl

Eindhoven University of Technology, The Netherlands

Abstract. Recent work by van Glabbeek and coauthors suggests that the liveness property for Peterson's mutual exclusion algorithm, which states that any process wanting to enter the critical section will eventually enter it, cannot be analysed in CCS and related formalisms. In our article, we explore the formal underpinning of this suggestion and its ramifications. In particular, we show that the liveness property for Peterson's algorithm *can* be established convincingly with the mCRL2 toolset, which has a conventional ACP-style process-algebra based specification formalism.

1 Introduction

A process-algebraic specification of a distributed algorithm or system typically includes unrealistic finite or infinite computations in which progress in some component halts. Their mere presence often sits in the way of a proof that the algorithm or system satisfies a set of desirable liveness properties. The go-to solution is to exclude these unrealistic computations from consideration by imposing additional assumptions such as *progress* and *fairness* (see [7] for a comprehensive overview).

For the analysis of so-called *fair schedulers*—of which Peterson's mutual exclusion algorithm is an example—one should, however, use such fairness assumptions cautiously, as fair schedulers themselves are intended to realise the very aspect of fairness in a system. In [6], Van Glabbeek and Höfner propose *justness* as a criterion that is just strong enough to exclude unrealistic computation of fair schedulers:

> "Once a transition is enabled that stems from a set of parallel components, one (or more) of these components eventually partake in a transition." [7]

The semantics of process calculi are usually defined by associating with every expression a labelled transition system. Thus, the semantic mapping is forgetful with respect to the notion of component, which is crucial for the definition of which computations are just. To facilitate reasoning about liveness properties of processes specified in process calculi while taking justness into account, two solutions are proposed in the literature: In [4], the definition of *just path* uses that the states of the labelled transition system associated with an expression of the calculus (by its structural operational semantics) are themselves expressions of the calculus; hence, these expressions reflect component information. In [5], the operational semantics is revised so that it yields a labelled transition system enriched with a concurrency relation that reflects component information; the notion of *just path* is then formulated referring to the concurrency relation.

The disadvantage of both the aforementioned definitions of just path is that they preclude the use of state-of-the-art verification technology for process calculi that has been developed over the past two decades relying firmly on the forgetful semantic mapping to labelled transition systems. Our aim, in [1], is to present a method by

which the mCRL2 toolset [2] can be used to verify that all just paths associated with an mCRL2 specification satisfy a liveness property, and to establish its correctness. Our method does not require an adaptation of the mCRL2 toolset itself. Instead, it relies on a disciplined use of labels in an mCRL2 process specification, by which from the label it can be inferred exactly which components contribute to the execution of the transition with that label. Then a justness assumption can be built into the μ-calculus formula expressing the liveness property. In the remainder of this article we first summarise the main ideas regarding the disciplined use of labels in the mCRL2 process specification, and then we discuss how to formalise liveness for all just paths in the μ-calculus.

2 Label-Based Justness for mCRL2

In mCRL2 a process is specified as a parallel composition of sequential components. The language has a flexible mechanism, inherited from ACP, to define interaction between those components. One can, e.g., specify that if one component can execute a transition labelled with a and another component can execute a transition labelled with b, then the two components may synchronise, resulting in a transition labelled with c. This is achieved by including the specification of a so-called *communication function*, expressing that labels a and b communicate to c. (In contrast, the language CCS has a fixed communication function that only allows transitions labelled with complementary labels (a and $\bar{a}$) to synchronise and the resulting transition is always labelled with the special label τ.) The flexibility to specify a communication function is crucial to our definition of justness, because it allows one to encode in the label c which components are contributing to the interaction it represents.

Our formalisation of the notion of *just path* in the context of an mCRL2 specification, which is inspired by the formalisation in [5], assumes that the following two mappings are defined:

1. *npc* associates with every label a set of *necessary participants*, i.e., components that participate in the interaction; and
2. *afc* associates with every label a set of *affected components*, i.e., components that are thought to be affected by the interaction.

The pair (npc, afc) is called a *component assignment* for the specification.

The distinction between necessary participant and affected component is important when, for example, modelling shared variables in process calculi. Since process calculi adhere strictly to the message passing paradigm, shared variables should be modelled as separate components. As pointed out in [4], to get a realistic notion of justness (e.g., for a model of Peterson's algorithm), the activity of reading the value of the variable should, however, not be treated as affecting the component representing the shared variable. To facilitate this, we partition the set of labels used in the mCRL2 specification into a set $\mathcal{S}$ of *signals* and a set $\mathcal{A}$ of *actions*. Transitions labelled by signals are special in that they should not change state; this is a property that needs to be established for the mCRL2 specification at hand. Moreover, the communication function should respect signals in the sense that it should not yield a signal, unless it was applied exclusively to signals.

For a correct definition of just path, it is important that the component assignment truly reflects the component structure; such a component assignment we shall call *consistent*. It is not possible to associate a consistent component assignment with every mCRL2 specification, but in [1] we give fairly liberal sufficient conditions that ensure that a consistent component assignment does indeed exist. Roughly, these conditions require that the sets of labels occurring in components are disjoint, that the component assignment assigns each such label to the correct

component, and that the communication function is consistent with the component assignment. It is worth reiterating that the flexible communication mechanism of mCRL2 is crucial for the latter. We refer to [1] for the formalities. It is also argued in [1], that Peterson's algorithm for mutual exclusion can be modelled by an mCRL2 specification for which there exists a consistent component assignment.

Finally, as argued in [5], justness is used to specify which paths represent a *complete computation* of the system. The distinction between *blocking* and *non-blocking* actions is relevant for determining when a computation is complete. Blocking actions are not entirely under the control of the specified system; their execution may depend on interaction with the environment. A non-blocking action is assumed to be completely under the control of the system. Complete computations may therefore only end in a state in which only blocking actions are enabled.

We now proceed to define the notion of *just path*. First, to define the notion of path we refer to the labelled transition system associated with an mCRL2 specification. A *path* in this transition system is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions, starting with a state and if it is finite also ending with a state, such that $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all relevant i. Furthermore, we say that an action a is *enabled* in a state s if there exists s' such that $s \xrightarrow{a} s'$. The component assignment for the mCRL2 specification induces a *concurrency relation* on labels: we define that $\lambda_1 \frown\!\!\!\!\bullet\ \lambda_2$ if, and only if, $npc(\lambda_1) \cap afc(\lambda_2) = \emptyset$. This concurrency relation is used to define the notion of just path.

Definition 1. *Let $\mathcal{B} \subseteq \mathcal{A}$ be a set of* blocking actions. *A path π is* $\mathcal{B}$-just *if for every action $a \notin \mathcal{B}$ that is enabled in some state s on π, an action a' occurs in the suffix of π starting at s such that $a \not\frown\!\!\!\!\bullet\ a'$.*

Indeed, the above formalisation of a just path captures the essence of the informal definition. For every action transition involving some set of parallel components that is enabled at some point on the path, ultimately some other transition is executed by a set of parallel components interfering with those enabling the first transition.

3 Off-the-Shelf Verification of Liveness

Due to the nature of justness, which 'dynamically' checks for enabledness of actions along a path, and their future elimination, it is not obvious whether one can express liveness properties restricted to just paths only, in a suitable modal logic. In spite of this dynamic nature, we show that the modal μ-calculus (supported by mCRL2) can be used to express typical liveness properties under justness.

As an illustration, Table 1 displays a template formula asserting (the violation of) the property **a**-**b**-liveness, stating that on all just paths, an action **a** is inevitably followed by action **b**. This template formula can be instantiated by a user wishing to carry out a liveness verification of an algorithm: it only requires information concerning which labels are designated as signals. As a result, mCRL2 can also be used to verify liveness properties of algorithms such as Peterson's.[1] As a bonus, counterexamples [3] can be provided in case of liveness violations.

Notice that the template formula asserts the *existence* of a just path violating the liveness property, which is conceptually simpler than the dual problem of asserting that the liveness property holds true on all just paths. A just path constitutes a violation to our liveness property exactly when (1) this path has a prefix leading to a state, reached by an **a**-labelled transition, and (2) along the just suffix of this path action **b** never takes place.

[1] The mCRL2 sources can be found in the *academic* example directory of the mCRL2 repository, see `https://github.com/mCRL2org/mCRL2`, revision `b45856d9a`.

Table 1. Template formula that characterises the set of states that admit a just path violating **a**-**b**-liveness. The user provides the sets $\mathcal{A}$ and $\overline{\mathcal{B}} = \mathcal{A} \setminus \mathcal{B}$, the relation $\smile\!\!\!\!\bullet$ and the pair of actions **a** and **b** to instantiate/generate the formula for checking a transition system associated with an mCRL2 specification with a consistent component assignment. Note that the modality $\langle\lambda\rangle\phi$ asserts that there is a λ-labelled transition leading to a state satisfying ϕ. The fixed points indicate that one is interested in the least (μ) or largest (ν) set of states satisfying the formula.

$$\mathsf{violate} = \quad \mu W.\ (\ \langle \mathbf{a} \rangle \mathsf{invariant} \vee \bigvee_{\lambda \in \mathcal{A}} \langle \lambda \rangle W\ )$$

$$\mathsf{invariant} = \nu Y.\ \bigwedge_{\lambda \in \overline{\mathcal{B}}} (\langle \lambda \rangle \top \Rightarrow \mathsf{elim}(\lambda))$$

$$\mathsf{elim}(\lambda) = \quad \mu Q.\ (\ \bigvee_{\lambda' \in \#\lambda \setminus \{\mathbf{b}\}} \langle \lambda' \rangle Y \vee \bigvee_{\lambda' \in \mathcal{A} \setminus (\#\lambda \cup \{\mathbf{b}\})} \langle \lambda' \rangle Q\ )$$

where $\#\lambda = \{\lambda' \mid \lambda \not\smile\!\!\!\!\bullet\ \lambda'\}$

Formula violate simply characterises the set of states satisfying property (1), i.e., those states admitting paths in which an **a**-action enters a state admitting a path satisfying property (2). The states that meet the latter property are represented by formula invariant, characterising exactly those states that allow for a **b**-free just path. The justness of that path is captured by the fact that, along that path, each enabled, non-blocking action λ is eliminated along that path. The latter is expressed by $\mathsf{elim}(\lambda)$, asserting that there is a finite **b**-free path consisting of actions that do not interfere with λ, and an action λ', which *does* interfere with λ, leads to a state again satisfying invariant. It is a property of the transition system associated with mCRL2 specifications with a consistent component assignment that on this finite path towards the action interfering with λ, all other enabled non-blocking actions remain enabled so long as no action interfering with them is executed.

Theorem 1. *For an mCRL2 specification with a consistent component assignment it holds that all just paths starting in state s satisfy* **a**-**b**-*liveness iff* $s \notin [\![\mathsf{violate}]\!]$.

References

1. Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. Off-the-shelf automated analysis of liveness properties for just paths. *Acta Informatica*, 57(3-5):551–590, 2020.
2. Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *TACAS (2)*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019.
3. Sjoerd Cranen, Bas Luttik, and Tim A. C. Willemse. Evidence for fixpoint logic. In *CSL*, volume 41 of *LIPIcs*, pages 78–93. Schloß Dagstuhl - Leibniz-Zentrum für Informatik, 2015.
4. Victor Dyseryn, Rob J. van Glabbeek, and Peter Höfner. Analysing mutual exclusion using process algebra with signals. In Kirstin Peters and Simone Tini, editors, *Proceedings EXPRESS/SOS 2017*, volume 255 of *EPTCS*, pages 18–34, 2017.
5. Rob J. van Glabbeek. Justness - A completeness criterion for capturing liveness properties (extended abstract). In *FoSSaCS*, volume 11425 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2019.
6. Rob J. van Glabbeek and Peter Höfner. 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.
7. Rob J. van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019.