

This is a repository copy of *Compositional and Local Livelock Analysis for CSP*.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/126283/>

Version: Accepted Version

---

**Article:**

Filho, M. S. Conserva, Oliveira, Marcel Vinicius Medeiros, Sampaio, A. C. A. et al. (1 more author) (2018) Compositional and Local Livelock Analysis for CSP. Information Processing Letters. ISSN 0020-0190

<https://doi.org/10.1016/j.ipl.2017.12.011>

---

**Reuse**

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing [eprints@whiterose.ac.uk](mailto:eprints@whiterose.ac.uk) including the URL of the record and the reason for the withdrawal request.

# Accepted Manuscript

Compositional and Local Livelock Analysis for CSP

M.S. Conserva Filho, M.V.M. Oliveira, A. Sampaio, Ana Cavalcanti

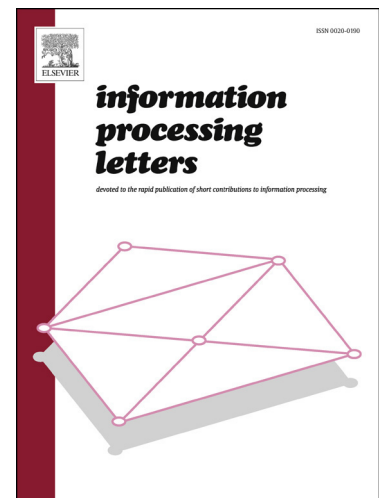
PII: S0020-0190(18)30003-6  
DOI: <https://doi.org/10.1016/j.ipl.2017.12.011>  
Reference: IPL 5622

To appear in: *Information Processing Letters*

Received date: 7 December 2016  
Revised date: 10 August 2017  
Accepted date: 30 December 2017

Please cite this article in press as: M.S. Conserva Filho et al., Compositional and Local Livelock Analysis for CSP, *Inf. Process. Lett.* (2018), <https://doi.org/10.1016/j.ipl.2017.12.011>

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.



## Highlights

- Livelock freedom analysis for CSP can scale using local and compositional techniques.
- The approach avoids the traditional explicit state-space exploration of the system.
- The strategy is based on a local analysis of the shortest event sequences (traces) that represent a recursive behaviour in the CSP model.
- We provide evidence of the efficiency of the proposed approach.

# Compositional and Local Livelock Analysis for CSP

M. S. Conserva Filho<sup>a</sup>, M. V. M. Oliveira<sup>a</sup>, A. Sampaio<sup>b</sup>, Ana Cavalcanti<sup>c</sup>

<sup>a</sup>*Universidade Federal do Rio Grande do Norte, Brazil*

<sup>b</sup>*Universidade Federal de Pernambuco, Brazil*

<sup>c</sup>*University of York, UK*

---

## Abstract

The success of component-based techniques for software construction relies on trust in the emergent behaviour of the compositions. Here, we propose an efficient correct-by-construction technique for building livelock-free CSP models. Its verification conditions are based on a local analysis of the shortest event sequences (traces) that represent a recursive behaviour in the CSP model. This affords significant gains in performance in model checking. We evaluate our strategy based on models of the Milner's scheduler and the dining philosophers.

*Keywords:* Process Algebra, Divergence, Model Checking, Components

---

## 1. Introduction

Compositional modelling and verification approaches are popular [4], but rely on trust in the emergent behaviour of the compositions. Process algebras are among the adopted formalisms. CSP [6, 10] is a well established process algebra to model and verify concurrent systems. CSP offers consolidated semantic models that support a wide range of verifications, including livelock freedom. A system is livelock free (divergence free) if there exists no state from which it internally computes through an infinite sequence of internal actions [10].

The main approach to prove divergence freedom requires a global analysis of the system. This strategy is automated for CSP, for instance, by FDR4 [5]. One alternative is a static analysis of the syntactic structure of a process [9]. For that, syntactic rules are proposed either to classify CSP systems as livelock-free or to report an inconclusive result. This approach is implemented in SLAP [9].

We present a technique based on a local analysis, in which we can identify livelock situations when compositions are being performed, predicting, by construction, global property based on known local properties of the components [1]. Our strategy aims at reducing complexity for verifying the absence of divergence, especially comparing with the approach in [9]. We illustrate our technique based on models of the Milner's scheduler and the dining philosophers, and show that it outperforms both FDR4 and SLAP. In cases in which livelock

---

*Email address:* `madiel@ppgsc.ufrn.br` (M. S. Conserva Filho)

21 freedom is not ensured, we either identify the possibility of divergence or report  
22 an inconclusive result. This incompleteness is the trade-off for scalability.

23 The next section briefly describes our evaluation strategy. Section 3 describes  
24 our technique, whose performance is evaluated in Section 4.

## 25 2. Material and methods

26 The demonstration of the usefulness and efficiency of our technique consists  
27 of a comparative analysis of three different scenarios: (i) the traditional global  
28 analysis of FDR4, (ii) the static livelock-analysis of SLAP, and (iii) our local  
29 livelock analysis, which is presented in the next section. We have developed two  
30 case studies: the Milner's task scheduler [7], which can be modelled as a ring of  
31 cells with pairwise synchronisation, and the dining philosophers [10]. All CSP  
32 scripts used in the case studies can be found at [goo.gl/mAZWXq](http://goo.gl/mAZWXq). We have used  
33 a server with 4 core AMD Phenom II, and 8 GB of RAM in a Ubuntu system.

## 34 3. Theory

35 In CSP, when composing divergence-free processes, divergent behaviour can  
36 arise from the use of hiding [10]. For a given CSP process  $P$  and a set of  
37 events  $X$ , the process  $P \setminus X$  converts visible occurrences of events of  $P$  in  $X$   
38 into internal events. This transformation may yield an infinite loop of internal  
39 events. For instance,  $P = (a \rightarrow P) \setminus \{a\}$  is defined in terms of the prefix  
40 operator ( $\rightarrow$ ): it engages in event  $a$  and then recurses, but it diverges because  
41 the event  $a$  is hidden, hence,  $P$  indefinitely performs internal events without  
42 communicating with its environment. If a process can engage in an unbroken  
43 sequence of events from a set  $X$ , we must ensure that  $X$  cannot be hidden.

44 The hiding operator is also implicitly used in a particular kind of parallel  
45 composition: the *linked parallel composition*  $P[a \leftrightarrow b]Q$ , in which  $P$  and  $Q$   
46 proceed in parallel with communications on  $a$  in  $P$  becoming hidden synchroni-  
47 sations with communications on  $b$  in  $Q$ . Communications on other channels are  
48 interleaved: they do not require synchronisation. In general, multiple channels  
49 may be linked as, for example, in  $P[a \leftrightarrow b, c \leftrightarrow d]Q$ .

50 We propose a constructive approach which guarantees that, for livelock-  
51 free processes that obey certain conditions and are composed pairwise using  
52 linked parallel, the resulting composition is livelock-free. To achieve scalability,  
53 we perform an optimisation (which we refer in Figure 1 as *OP*) that prunes  
54 the alternative behaviours of the resulting composition with interleaved events,  
55 choosing only one of the alternatives.

56 Our approach is based on three main verifications, which are systematically  
57 applied (see Figure 1): the Simple Verification (*SV*) ensures livelock freedom  
58 based on an individual analysis of the processes involved in the composition.  
59 The absence of livelock is guaranteed if one of the processes is livelock-free after  
60 hiding its linking events locally. If that fails, the Complex Verification (*CV*)  
61 checks if the linked processes are able to communicate in an infinite loop via

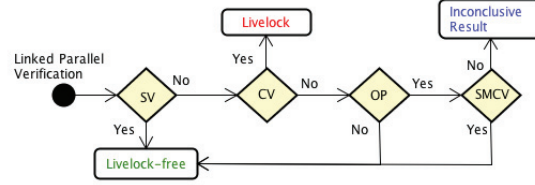


Figure 1: BPM Model of the Livelock Analysis for Linked Parallel Composition

the linked (internal) events. If they are, we have a livelock. Otherwise, if the optimisation (*OP*) has not been applied, the composition is livelock-free. If, however, the optimisation has been applied, our strategy guarantees livelock freedom only if we have a Safe Multiple Composition (*SMCV*), which does not link events on a many-to-many fashion. Otherwise, the interleaved events pruned by our optimisation may lead the system to divergence. Our strategy is, therefore, inconclusive in such cases. In what follows, we present the basic definitions used in our technique and formally describe these local verifications.

### 3.1. Basic Definitions

Our method considers developments that use livelock-free basic processes, which can be described using most of the CSP main operators, including conditionals, tail and mutual recursions. We also consider parameters. Further information on basic processes can be found in [3]. Parallelism (and hiding) is achieved by composing processes (either basic or resulting from previous compositions) using the linked parallel composition.

The first step of our technique is to identify the infinite behaviours of a given process. For that, we use a pair  $(tr, mip)$  of sequences (traces). Its first element is a trace that leads a given process to a recursive behaviour. The second one is a minimal interaction pattern of a given process, that is, the shortest finite sequence of events that represents the recursion itself. The set  $XIP(P)$  contains all possible pairs  $(tr, mip)$  of the process  $P$ .

To exemplify our method, we introduce a classical concurrent system, the dining philosophers [10]. It consists of philosophers sitting at a round table that need to acquire a pair of shared forks before eating. The behaviour of each philosopher and each fork is modelled as a process  $P_i$  or  $F_i$  for values  $i$  from a set  $ID$  of philosopher and fork identifiers. We consider two philosophers and two forks and use  $ID = \{1, 2\}$ . A channel  $fk : ID.ID.EV$ , where  $EV = \{U, D\}$  defines events  $fk.i.j.e$  that indicate that the fork  $i$  is put up or down, depending on whether  $e$  is  $U$  or  $D$ , by the philosopher  $j$ . The fork processes are as follows.

$$\begin{aligned}
 F_1 &= fk.1.1.U \rightarrow fk.1.1.D \rightarrow F_1 \sqcap fk.1.2.U \rightarrow fk.1.2.D \rightarrow F_1 \\
 F_2 &= fk.2.2.U \rightarrow fk.2.2.D \rightarrow F_2 \sqcap fk.2.1.U \rightarrow fk.2.1.D \rightarrow F_2
 \end{aligned}$$

Initially, a fork can be picked up by either philosopher. Once it is picked up, it can only be put down by the same philosopher. Accordingly, the process

93  $F_1$  offers a deterministic choice ( $\square$ ): it engages either on the events  $fk.1.1.U$   
 94 or  $fk.1.2.U$ . The prefix operator ( $\rightarrow$ ) states that the corresponding down  
 95 event ( $D$ ) is offered afterwards. The process recurses after the down event.  
 96 Hence,  $XIP(F_1) = \{(\langle \rangle, \langle fk.1.1.U, fk.1.1.D \rangle), (\langle \rangle, \langle fk.1.2.U, fk.1.2.D \rangle)\}$ . In this  
 97 example, as  $F_1$  returns to its initial state,  $tr$  is the empty trace ( $\langle \rangle$ ).

98 Similarly,  $pfk.j.i.e$  records the action  $e$  on fork  $j$  by philosopher  $i$ . The  
 99 channel  $wk : ID$  defines events  $wk.i$ , indicating that the philosopher  $i$  has just  
 100 woken up. Finally, the channel  $lf : ID.LF$ , where  $LF = \{T, E\}$  defines events  
 101  $lf.i.l$ , indicating that the philosopher  $i$  is either thinking ( $T$ ) or eating ( $E$ ).

$$\begin{aligned} P_1 &= wk.1 \rightarrow PS_1 \\ PS_1 &= lf.1.T \rightarrow pfk.1.1.U \rightarrow pfk.2.1.U \rightarrow lf.1.E \rightarrow pfk.1.1.D \rightarrow \\ &\quad pfk.2.1.D \rightarrow PS_1 \\ P_2 &= wk.2 \rightarrow PS_2 \\ PS_2 &= lf.2.T \rightarrow pfk.1.2.U \rightarrow pfk.2.2.U \rightarrow lf.2.E \rightarrow pfk.1.2.D \rightarrow \\ &\quad pfk.2.2.D \rightarrow PS_2 \end{aligned}$$

102 The process  $P_1$  initially performs the event  $wk.1$  and then behaves as  $PS_1$ ,  
 103 which represents the recursive behaviour of the philosopher: before eating, he  
 104 thinks and picks the forks up; after eating, he puts the forks down. In this case,  
 105  $XIP(P_1) = \{(\langle wk.1 \rangle, \langle lf.1.T, pfk.1.1.U, pfk.2.1.U, lf.1.E, pfk.1.1.D, pfk.2.1.D \rangle)\}$ .

106 We are now able to calculate which events of a given process can be hidden  
 107 without introducing livelock. The function  $Allowed(P)$  identifies all sets of  
 108 events that can be individually hidden from  $P$ . Here,  $\Sigma$  is the set of all possible  
 109 events,  $MIP(P)$  is the set that contains only the second element of the pairs  
 110  $(tr, mip)$  in  $XIP(P)$ , and  $\text{ran}(s)$  is the set of the elements of the sequence  $s$ .

111 **Definition 3.1 (Allowed).** Let  $P$  be a livelock-free CSP process. The set of  
 112 sets of events of  $P$  that can be hidden with no introduction of divergence is given  
 113 by  $Allowed(P)$ , which is defined as follows:

$$Allowed(P) = \{cs : \mathbb{P}\Sigma \mid \neg \exists s : MIP(P) \bullet \text{ran}(s) \subseteq cs\}$$

114 In our example, hiding either  $\{fk.1.1.U, fk.1.1.D\}$  or  $\{fk.1.2.U, fk.1.2.D\}$  from  
 115  $F_1$  introduces divergence because there exists an element in  $MIP(F_1)$  that only  
 116 has events in such sets; they are not in  $Allowed(F_1)$ . Our concern here is only  
 117 with the sequences in  $MIP(P)$ , since livelock may be introduced if we hide all  
 118 elements of a sequence that is recursively offered by  $P$ . The first element of the  
 119 pair  $(tr, mip)$  is not relevant in this context because livelock is never introduced  
 120 if we hide all elements of a sequence that is offered a finite number of times. We  
 121 are now able to formally define our local verifications, as illustrated in Figure 1.

### 122 3.2. Simple Verification

123 As an example, we consider  $PComp_1 = P_1[pfk.1.1 \leftrightarrow fk.1.1]F_1$ , which is  
 124 equivalent to  $P_1[pfk.1.1.U \leftrightarrow fk.1.1.U, pfk.1.1.D \leftrightarrow fk.1.1.D]F_1$ . We observe  
 125 that  $\{pfk.1.1.U, pfk.1.1.D\}$  is in  $Allowed(P_1)$  and  $\{fk.1.1.U, fk.1.1.D\}$  is not  
 126 in  $Allowed(F_1)$ . Nevertheless, the composition is livelock-free because, after

synchronisation on *pfk* events,  $P_1$  necessarily has to engage on an independent visible event, such as *lf.1.E*. We present below our first result, which justifies our claim in this example. Here,  $\alpha(P)$  is the set of events that  $P$  can communicate.

**Proposition 3.1 (SV).** *Let  $P$  and  $Q$  be two livelock-free CSP processes with  $\alpha(P) \cap \alpha(Q) = \emptyset$ , and  $I = \{i_1, \dots, i_n\}$  and  $O = \{o_1, \dots, o_n\}$  two disjoint sets of events ( $I \cap O = \emptyset$ ). If either  $I \in \text{Allowed}(P)$  or  $O \in \text{Allowed}(Q)$ , then the composition  $P[i_1 \leftrightarrow o_1, \dots, i_n \leftrightarrow o_n]Q$  is livelock free.*

This proposition states that, if any of the connecting sets of events used in the composition belongs to the set of *Allowed* events of the corresponding process, the linked parallel composition is livelock-free.

### 3.3. Complex Verification

If the restriction indicated in Proposition 3.1 does not hold, we have local possibilities of livelock. This, however, does not necessarily introduce livelock because the composition diverges only if both processes synchronise indefinitely on the composed events. As an example, we consider the following processes.

$$\begin{array}{ll} P_3 = a \rightarrow P_4 & Q_3 = e \rightarrow Q_4 \\ P_4 = b \rightarrow c \rightarrow P_4 & Q_4 = f \rightarrow Q_3 \end{array}$$

Here, we have  $XIP(P_3) = \{(\langle a \rangle, \langle b, c \rangle)\}$  and  $XIP(Q_3) = \{(\langle e \rangle, \langle e, f \rangle)\}$ . Neither  $\{b, c\}$  is in  $\text{Allowed}(P_3)$  nor  $\{e, f\}$  is in  $\text{Allowed}(Q_3)$ . Therefore, if we hide the set of events  $\{b, c\}$  in  $P_3$ , livelock is introduced. The same takes place when we hide  $\{e, f\}$  in  $Q_3$ . However, if we perform the composition  $P_3[b \leftrightarrow f, c \leftrightarrow e]Q_3$ , livelock is not introduced because we have a deadlock.

To make this verification, we consider  $\text{ProjXIP}(P, cs)$ , which identifies the pairs  $(tr, mip)$  in  $XIP(P)$  in which  $mip$  has only elements in  $cs$ . With  $cs$  as the set of events hidden in a composition of processes  $P$  and  $Q$ , we identify the sequences that may cause livelock using  $\text{ProjXIP}(P, cs)$  and  $\text{ProjXIP}(Q, cs)$  as described next. Since the elements that are not in  $cs$  do not contribute to the synchronisations, they are removed from  $tr$  in the pairs defined by  $\text{ProjXIP}$ .

To check for the possibility of (indefinite) synchronisation between parallel processes, we compare their sets of pairs defined by  $\text{ProjXIP}$  and identify the possibility of matching communications on the linked events. Since these are (potentially) different events, like  $b$  and  $c$ , and  $e$  and  $f$  in the example above, we rename the pairs of traces in  $\text{ProjXIP}(P)$  using the function  $\text{RenXIP}(P, f)$ . Nevertheless, only using  $\text{RenXIP}$  is not enough to compare the elements of the pairs. As an example, we consider the following CSP processes.

$$\begin{array}{ll} P_5 = a.1 \rightarrow P_6 & Q_5 = b.1 \rightarrow Q_6 \\ P_6 = a.2 \rightarrow a.1 \rightarrow P_6 & Q_6 = c.1 \rightarrow c.2 \rightarrow Q_6 \end{array}$$

Here, we have  $\text{ProjXIP}(P_5, \{a\}) = \{(\langle a.1 \rangle, \langle a.2, a.1 \rangle)\}$  and  $\text{ProjXIP}(Q_5, \{c\}) = \{(\langle \rangle, \langle c.1, c.2 \rangle)\}$ . We use renaming functions  $f_1 = \{a.1 \mapsto x1, a.2 \mapsto x2\}$  and  $f_2 = \{c.1 \mapsto x1, c.2 \mapsto x2\}$  so that the linked events in  $P_5[a \leftrightarrow c]Q_5$  are renamed



163 to the same fresh events  $x1$  and  $x2$ . The choice of names  $x1$  and  $x2$  is arbitrary.  
 164 With these renaming functions, we have  $RenXIP(P_5, f_1) = \{(\langle x1 \rangle, \langle x2, x1 \rangle)\}$   
 165 and  $RenXIP(Q_5, f_2) = \{(\langle \rangle, \langle x1, x2 \rangle)\}$ .

166 Renaming the projected pairs is still not enough to identify the matching  
 167 in these traces directly. In this case, before the recursion, the trace in  $tr$  of  
 168  $RenXIP(P_5, f_1)$  synchronises with the first element in  $mip$  of  $RenXIP(Q_5, f_2)$ .  
 169 After that, an infinite loop is reached due to the synchronisation of the  $mip$   
 170  $\langle x2, x1 \rangle$  of  $RenXIP(P_5, f_1)$  with  $\langle x2, x1 \rangle$ , which is other possible behaviour in  
 171 which the loop can be observed in  $RenXIP(Q_5, f_2)$ . That is, besides the origi-  
 172 nal pair in  $RenXIP(Q_5, f_2)$ , we also can observe the recursion through the pair  
 173  $(\langle x1 \rangle, \langle x2, x1 \rangle)$ . For that, we consider  $RenXIP^+$ , which identifies all pairs ob-  
 174 tained from those in  $RenXIP$  that lead to a loop. They are possibilities in which  
 175 the original pairs can perform the loops. With this, we identify that  $P_5$  and  $Q_5$   
 176 communicate continuously via internal synchronisations on  $a$  and  $c$ .

177 Our strategy uses these enriched sets to identify a *Minimum Common In-*  
 178 *teraction Pattern (MCIP)* because we only need to perform this verification  
 179 until the first minimum sequence is found; it identifies the first trace that leads  
 180 the composition to divergence. The function  $MCIP(S_1, S_2)$  applies to two en-  
 181 riched sets of projected renamed pairs,  $S_1$  and  $S_2$ , and identifies the commom  
 182 sequences that can be reached by the concatenation of the elements of  $tr$  with  
 183 the arbitrary concatenation of the elements of  $mip$  of both sets. In our example,  
 184 the minimum commom sequence is  $\langle x1, x2, x1 \rangle$ .

185 We now present our second main result for ensuring the absence of divergence  
 186 for non-trivial linked parallel compositions.

187 **Proposition 3.2 (CV).** *Let  $P$  and  $Q$  be two livelock-free CSP processes with*  
 188  *$\alpha(P) \cap \alpha(Q) = \emptyset$ ,  $I = \{i_1, \dots, i_n\}$  and  $O = \{o_1, \dots, o_n\}$  two disjoint sets of events,*  
 189  *$X = \{x_1, \dots, x_n\}$  a set of fresh event names, and  $f_1 = \{i_1 \mapsto x_1, \dots, i_n \mapsto x_n\}$  and*  
 190  *$f_2 = \{o_1 \mapsto x_1, \dots, o_n \mapsto x_n\}$  two renaming functions from events to fresh event*  
 191 *names. If  $MCIP(RenXIP(P, f_1)^+, RenXIP(Q, f_2)^+) = \emptyset$ , then the composition*  
 192  *$P[i_1 \leftrightarrow o_1, \dots, i_n \leftrightarrow o_n]Q$  is livelock-free; otherwise, there is a livelock.*

193 Proposition 3.2 states that livelock is not introduced if there exists no com-  
 194 mon sequence that can be reached by the concatenation of the elements of any  
 195 enriched renamed projected pairs of the processes involved in the composition.  
 196 Otherwise, besides indicating the possibility of identifying livelock compositions,  
 197 we also capture the traces that lead the composition to divergence.

198 Although the method so far is complete, it does not scale for complex com-  
 199 positions. We, therefore, consider an optimisation that prunes the alternative  
 200 behaviours induced by the parallelism. With this, we lose completeness and  
 201 need to consider a more elaborate strategy, but this is the trade-off for scala-  
 202 bility. If the optimisation has been performed, the verification is based on the  
 203 identification of a specific pattern of composition, as discussed next.

#### 204 3.4. Safe Multiple Composition Verification

205 In CV, besides analysing the synchronisation of the processes, we also have to  
 206 take into account the possible combinations of independent (interleaved) events

that can be performed after a parallel composition. As an example, we consider the following livelock-free CSP processes.

$$P_7 = a \rightarrow b \rightarrow P_7 \square c \rightarrow P_7 \quad Q_7 = d \rightarrow e \rightarrow Q_7 \quad R_7 = f \rightarrow g \rightarrow R_7$$

After synchronising on  $a$  and  $d$ , the composition  $PQ_7 = P_7[a \leftrightarrow d]Q_7$  needs to engage both in  $b$  and in  $e$  before it recurses. This can happen in two different ways:  $\langle b, e \rangle$  or  $\langle e, b \rangle$ . In general, we have an interleaving on events that do not require synchronisation, and, from a practical point of view, the consideration of these traces can lead to an explosion on the number of possible behaviours. To make our strategy scalable, we consider just one of the traces that can arise from the interleaving. As a result, we have,  $XIP(PQ_7) = \{(\langle \rangle, \langle b, e \rangle), (\langle \rangle, \langle c \rangle)\}$ . The analysis of a further composition of  $PQ_7$  may be impacted by this. For example, in  $PQR_7 = PQ_7[b \leftrightarrow g, e \leftrightarrow f]R_7$ , there is no divergence, according to our strategy as presented so far; however, if we had considered the pair  $(\langle \rangle, \langle e, b \rangle)$  as part of  $XIP(PQ_7)$ , then our strategy would identify a divergence that indeed exists. The optimisation may cause the livelock analysis to fail.

This problem can be circumvented by imposing restrictions on the composition. Our strategy requires that, in every composition, each basic process on the left-hand side is linked with just one basic process on the right-hand side, and vice-versa. The verification of this requirement uses the notion of *Basic Process Alphabet* ( $BPA(P)$ ): a set that contains the alphabets of the basic processes of a given process  $P$ . Each element of  $BPA(P)$  is the alphabet of a distinct basic process of  $P$ . In our example, we have:

$$BPA(P_7) = \{\{a, b, c\}\} \quad BPA(Q_7) = \{\{d, e\}\} \quad BPA(R_7) = \{\{f, g\}\}$$

The resulting  $BPA$  of a composition is the union of the  $BPA$ s of the processes involved in the composition with the linked events removed from them. For example,  $BPA(PQ_7) = \{\{b, c\}, \{e\}\}$ .

The analysis of compositions that only connect basic processes is not affected by our optimisation. This is because in our optimised verification, we still consider all pairs of basic processes. It is the compositions of composed processes that are affected, that is, compositions that originate different traces that always communicate on the same events. As an example, we consider  $PQR_7$  presented above. It does not satisfy our restriction because the left linked events  $b$  and  $e$  are originated from different basic processes in  $PQ_7$  ( $b \in \alpha(P_7)$  and  $e \in \alpha(Q_7)$ ). On the other hand,  $PQR_8 = PQ_7[b \leftrightarrow f, c \leftrightarrow g]R_7$  satisfies our restriction because  $R_7$  is a basic process and the composition only connects events from  $P_7$ , which is also a basic process in  $PQ_7$ . For such compositions, the search for an  $MCIP$  is correctly performed since all traces that may lead the composition to divergence are verified because the traces of basic processes are not optimised; they do not have parallel composition in their behaviours.

Our restriction, however, also allows connections of an arbitrary number of basic processes as long as they are effectively one-to-one connections. This condition is formally defined below. The expression  $R(S)$  is the relational image of the relation  $R : X \leftrightarrow Y$  on the set  $S \subseteq X$ .

**Definition 3.2 (Multiple Basic Processes Composition).** Let  $P$  and  $Q$  be two livelock-free CSP processes with  $\alpha(P) \cap \alpha(Q) = \emptyset$ , and  $I = \{i_1, \dots, i_n\}$  and  $O = \{o_1, \dots, o_n\}$  two disjoint sets of events ( $I \cap O = \emptyset$ ). Then, the composition  $P[i_1 \leftrightarrow o_1, \dots, i_n \leftrightarrow o_n]Q$  is a Multiple Basic Processes Composition if:

$$\begin{aligned} & MBPC(P, Q) \wedge MBPC(Q, P), \text{ where} \\ & MBPC(X, Y) = \\ & \quad \forall p : BPA(X) \bullet \\ & \quad \neg \exists q_1, q_2 : BPA(Y) \mid q_1 \neq q_2 \bullet q_1 \cap L(p) \neq \emptyset \wedge q_2 \cap L(p) \neq \emptyset \\ & \text{where } L = \{(i_1, o_1), \dots, (i_n, o_n)\}. \end{aligned}$$

This condition requires that for every BPA of  $P$ , there exists at most one BPA of  $Q$  that is being linked to it, and vice-versa.

Finally, we present our result for ensuring livelock-free linked parallel composition for cases in which an optimisation has been performed.

**Proposition 3.3 (SMCV).** Let  $P$  and  $Q$  be two livelock-free CSP processes with  $\alpha(P) \cap \alpha(Q) = \emptyset$ ,  $I = \{i_1, \dots, i_n\}$  and  $O = \{o_1, \dots, o_n\}$ , two disjoint sets of events ( $I \cap O = \emptyset$ ),  $X = \{x_1, \dots, x_n\}$  a set of fresh event names, and  $f_1 = \{i_1 \mapsto x_1, \dots, i_n \mapsto x_n\}$  and  $f_2 = \{o_1 \mapsto x_1, \dots, o_n \mapsto x_n\}$  two renaming functions from events to fresh event names. If the linked parallel composition  $P[i_1 \leftrightarrow o_1, \dots, i_n \leftrightarrow o_n]Q$  is a Multiple Basic Processes Composition and  $MCIP(RenXIP(P, f_1)^+, RenXIP(Q, f_2)^+) = \emptyset$ , then the linked parallel composition  $P[i_1 \leftrightarrow o_1, \dots, i_n \leftrightarrow o_n]Q$  is livelock free.

Proposition 3.3 states that a linked parallel composition is livelock-free in cases in which we do not have communications of basic processes on a many-to-many fashion and there exists no MCIP. Otherwise, our strategy is inconclusive.

We have implemented an algorithm that supports livelock verification using these concepts. Further details can be found elsewhere [3].

## 4. Results and Discussion

The comparative analysis to evaluate our strategy has been conducted for a livelock-free Milner's scheduler system and for a dining philosopher system. Table 1 and Table 2 summarise our results, where  $N$  is the size of the configuration of these systems (for instance, on the first example it is the number of cells and in the second the number of philosophers and forks), and  $\#$  represents the number of compositions. Furthermore, time is in seconds, \* indicates one hour timeout, and \*\* indicates memory overflow.

N	#	FDR4	SLAP	CLLA
10	9	1.68	0.39	0.72
100	99	**	**	1.98
1,000	999	**	**	7.75
2,000	1,999	**	**	12.72

Table 1: Results for the Milner's Scheduler

N	#	FDR4	SLAP	CLLA
10	19	**	19.72	1.49
100	199	**	*	4.21
1,000	1,999	**	**	53.40
10,000	10,999	**	**	3451.02

Table 2: Results for the Dining Philosopher

The results show that FDR4 and SLAP are unable to deal with large synchronous models. On the other hand, our method (CLLA) verified, for instance, the absence of divergence for 10,000 philosophers and 10,000 forks (20,000 CSP processes and 10,999 linked parallel compositions) in less than 58 minutes. This is a promising result in dealing with large and complex systems.

In [10], a technique called the order rule is proposed to check the absence of livelock. In summary, a network is proved to be livelock-free if there is a specific order on its components such that no component can communicate exclusively and infinitely with components lower than it in this order. This strategy has not been implemented so far, and, consequently, no practical experiment was provided in this work. Our strategy is not restricted to this communication pattern and analyses the components pairwise to improve performance.

Another classical and extremely relevant property in concurrent systems is deadlock freedom. Approaches to local and compositional deadlock analysis have gained significant attention in the literature, including, for instance [2, 8]. As in the case of livelock, the approaches are efficient, but incomplete.

We plan to extend our technique to consider other kinds of parallel composition. Of course, the impact in efficiency of these improvements would need to be analysed. Additional case studies are also in our research agenda.

## References

- [1] M. Abadi and L. Lamport. Composing specifications. *TOPLAS*, 15:73–132, 1993.
- [2] P. Antonino, T. Gibson-Robinson, and A.W. Roscoe. Efficient Deadlock-Freedom Checking Using Local Analysis and SAT Solving. In *IFM*, pages 345–360. Springer, 2016.
- [3] M. Conserva, M. Oliveira, A. Sampaio, and Ana Cavalcanti. Compositional and Local Livelock Analysis for CSP. Technical report, UFRN, 2017. <http://goo.gl/mAZWXq>.
- [4] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *ICSEA*. IEEE, 2006.
- [5] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *TACAS*, pages 187–201, 2014.
- [6] C. A. R. Hoare. Communicating sequential processes. *ACM*, 1978.
- [7] R. Milner. *Communication and concurrency*. Prentice hall, 1989.
- [8] M.V.M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, and A.W. Roscoe. Rigorous development of component-based systems using component metadata and patterns. *FAC*, pages 1–68.
- [9] J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. A static analysis framework for livelock freedom in CSP. *LMCS*, 2013.
- [10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.