Static Livelock Analysis in CSP*

Joël Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell

Department of Computer Science, Oxford University, UK
{joel,hrip,awr,jbw}@cs.ox.ac.uk

Abstract. In a process algebra with hiding and recursion it is possible to create processes which compute internally without ever communicating with their environment. Such processes are said to diverge or livelock. In this paper we show how it is possible to conservatively classify processes as livelock-free through a static analysis of their syntax. In particular, we present a collection of rules, based on the inductive structure of terms, which guarantee livelock-freedom of the denoted process. This gives rise to an algorithm which conservatively flags processes that can potentially livelock. We illustrate our approach by applying both BDD-based and SAT-based implementations of our algorithm to a range of benchmarks, and show that our technique in general substantially outperforms the model checker FDR whilst exhibiting a low rate of inconclusive results.

1 Introduction

It is standard in process algebra to distinguish between the visible and invisible (or silent) actions of a process. The latter correspond to state changes arising from internal computation, and their occurrence is not detectable or controllable by the environment. A process is said to *diverge* or *livelock* if it reaches a state from which it may forever compute internally through an infinite sequence of invisible actions. This is usually a highly undesirable feature of the process, described in the literature as "even worse than deadlock" [6, page 156]. Livelock invalidates certain analysis methodologies, and is often symptomatic of a bug in the modelling. However the possibility of writing down divergent processes arises from the presence of two crucial constructs, recursion and hiding. The latter converts visible actions into invisible ones and is a key device for abstraction.

We distinguish two ways in which a process may livelock. In the first, a process may be able to communicate an infinite unbroken sequence of some visible event, and this process then occurs inside the scope of an operator which hides that event. Alternatively, a process may livelock owing to the presence of an unguarded recursion. Roughly speaking, the latter means that the process may recurse without first communicating a visible action.

This paper is concerned with the problem of determining whether a process may livelock in the context of the process algebra CSP, although the principles upon which our analysis is based should be transferable to other process algebras

* A full version of this paper, including all proofs, is available as [11].

as well. While it is straightforward to show that the problem is in general undecidable[1], we are still able to provide a conservative (i.e., sound but incomplete) method of checking for the possibility of livelock: this method either correctly asserts that a given process is livelock-free, or is inconclusive. The algorithm is based on a static analysis[2] of the given process, principally in terms of the interaction of hiding, renaming, and recursion. This analysis naturally divides into two parts according to the two sources of livelock outlined above.

The basic intuitions underlying our approach are fairly straightforward. In part they mirror the guardedness requirements which ensure that well-behaved CSP process equations have unique, livelock-free fixed points [13, Chap. 8]. However, we extend the treatment of [13] by allowing guarded recursions to include instances of the hiding operator. Incidentally, Milner's notion of guarded recursions in CCS is similarly restricted by the requirement that variables not occur inside parallel compositions [9]. Complications arise mainly because we want to be able to fully incorporate hiding and renaming in our treatment, both of which can have subtle indirect effects on guardedness.

We note that the idea of guarded recursions is standard in process algebra. For instance, in Milner's framework, a variable is 'strongly guarded' in a given term if every free occurrence of the variable in the term occurs within the scope of a prefixing operator [9]. This notion is introduced in order to justify certain proof principles, such as that guaranteeing the uniqueness of fixed points up to bisimilarity. Strong guardedness has also been extended to a calculus with hiding and action refinement [2]. A key difference between our approach and these notions is that we seek to guarantee livelock-freedom, rather than merely the existence of unique fixed points.

In fact, there are few papers which deal with the problem of guaranteeing livelock-freedom in the setting of concurrent process calculi.[3] The existing work on livelock-freedom has mostly been carried out in the context of mobile calculi. [15] presents an approach for guaranteeing livelock-freedom for a certain fragment of the π-calculus. Unlike the combinatorial treatment presented here, this approach makes use of the rich theory of types of the π-calculus, and in particular the technique of logical relations. Another study of divergence-freedom in the π-calculus appears in [20], and uses the notions of graph types.

Note that CSP is predicated upon *synchronous* (i.e., handshake) communication. In terms of livelock analysis, different issues (and additional difficulties) arise in an asynchronous context (assuming unbounded communication buffers); see, e.g., [7, 8].

Of course, one way to check a process for divergence is to search for reachable cycles of silent actions in its state space, which is a labelled transition system built from the operational semantics. Assuming this graph is finite, this

[1] For example, CSP can encode counters, and is therefore Turing-powerful.

[2] Here *static analysis* is used to distinguish our approach from the state-space exploration methods that underlie model checking or refinement checking.

[3] In contrast, there are numerous works treating termination for the λ-calculus or combinatory logic [5, 10, 4].

can be achieved by calculating its strongly connected components. The latter can be carried out in time linear in the size of the graph, which may however be exponential (or worse) in the syntactic size of the term describing the process. By circumventing the state-space exploration, we obtain a static analysis algorithm which in practice tends to substantially outperform state-of-the-art model-checking tools such as FDR—see Sect. 6 for experimental comparisons.

Naturally, there is a trade-off between the speed and accuracy of livelock checking. It is not hard to write down processes which are livelock-free but which our analysis indicates as potentially divergent. However, when modelling systems in practice, it makes sense to try to check for livelock-freedom using a simple and highly economical static analysis before invoking computationally expensive state-space exploration algorithms. Indeed, as Roscoe [13, page 208] points out, the calculations required to determine if a process diverges are significantly more costly than those for deciding other aspects of refinement, and it is advantageous to avoid these calculations if at all possible.

Recent works in which CSP livelock-freedom plays a key role include [3] as well as [17, 16]; see also references within.

2 CSP: Syntax and Conventions

Let Σ be a finite set of *events*, with $\checkmark \notin \Sigma$. We write $\Sigma^\checkmark$ to denote $\Sigma \cup \{\checkmark\}$ and $\Sigma^{*\checkmark}$ to denote the set of finite sequences of elements from Σ which may end with $\checkmark$. In the notation below, we have $a \in \Sigma$ and $A \subseteq \Sigma$. R denotes a binary (renaming) relation on Σ. Its lifting to $\Sigma^\checkmark$ is understood to relate $\checkmark$ to itself. The variable X is drawn from a fixed infinite set of process variables.

CSP terms are constructed according to the following grammar:

$$P ::= STOP \mid a \longrightarrow P \mid SKIP \mid P_1 \sqcap P_2 \mid P_1 \,\Box\, P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P_1 \fatsemi P_2 \mid P \setminus A \mid P[R] \mid X \mid \mu X \bullet P \mid DIV \ .$$

$STOP$ is the deadlocked process. The prefixed process $a \longrightarrow P$ initially offers to engage in the event a, and subsequently behaves like P. $SKIP$ represents successful termination, and is willing to communicate $\checkmark$ at any time. $P \,\Box\, Q$ denotes the external choice of P and Q, whereas $P \sqcap Q$ denotes the internal (or nondeterministic) alternative. The distinction is orthogonal to our concerns, and indeed both choice operators behave identically over our denotational model. The parallel composition $P_1 \underset{A}{\parallel} P_2$ requires P_1 and P_2 to synchronise on all events in A, and to behave independently of each other with respect to all other events. $P \fatsemi Q$ is the sequential composition of P and Q: it denotes a process which behaves like P until P chooses to terminate (silently), at which point the process seamlessly starts to behave like Q. $P \setminus A$ is a process which behaves like P but with all communications in the set A hidden. The renamed process $P[R]$ derives its behaviours from those of P in that, whenever P can perform an event a, $P[R]$ can engage in any event b such that $a \; R \; b$. To understand the meaning of $\mu X \bullet P$, consider the equation $X = P$, in terms of the unknown X. While this

equation may have several solutions, it always has a unique least[4] such, written $\mu X \,.\, P$. Moreover, as it turns out, if $\mu X \,.\, P$ is livelock-free then the equation $X = P$ has no other solutions. Lastly, the process DIV represents livelock, i.e., a process caught in an infinite loop of silent events.

A CSP term is *closed* if every occurrence of a variable X in it occurs within the scope of a μX operator; we refer to such terms as *processes*.

Let us state a few conventions. When hiding a single event a, we write $P \setminus a$ rather than $P \setminus \{a\}$. The binding scope of the μX operator extends as far to the right as possible. We also often express recursions by means of the equational notation $X = P$, rather than the functional $\mu X \,.\, P$.

Let us also remark that CSP processes are often defined via *vectors* of mutually recursive equations. These can always be converted to our present syntax, thanks to Bekič's theorem [19, Chap. 10]. Accordingly, we shall freely make use of the vectorised notation in this paper.

3 Operational and Denotational Semantics

We present congruent (equivalent) operational and denotational semantics for CSP. For reasons of space, many details and clauses are omitted; a full account can be found in [11]. An extensive treatment of a variety of different CSP models can also be found in [13, 14]. The semantics presented below only distill those ideas from [13, 14] which are relevant in our setting.

The operational semantics is presented as a list of inference rules in SOS form; we only give below rules for prefixing, recursion, parallel composition, and hiding. In what follows, a stands for a visible event, i.e., belongs to $\Sigma^\checkmark$. $A \subseteq \Sigma$ and $A^\checkmark = A \cup \{\checkmark\}$. γ can be a visible event or a silent one ($\gamma \in \Sigma^\checkmark \cup \{\tau\}$). $P \xrightarrow{\gamma} P'$ means that P can perform an immediate and instantaneous γ-transition, and subsequently become P' (communicating γ in the process if γ is a visible event). If P is a term with a single free variable X and Q is a process, $[Q/X]P$ represents the process P with Q substituted for every free occurrence of X.

$$\frac{}{(a \longrightarrow P) \xrightarrow{a} P} \qquad \frac{}{\mu X \,.\, P \xrightarrow{\tau} [(\mu X \,.\, P)/X]P}$$

$$\frac{P_1 \xrightarrow{\gamma} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\gamma} P_1' \underset{A}{\parallel} P_2} \; [\gamma \notin A^\checkmark] \qquad \frac{P_2 \xrightarrow{\gamma} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\gamma} P_1 \underset{A}{\parallel} P_2'} \; [\gamma \notin A^\checkmark]$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2'} \; [a \in A^\checkmark]$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \; [a \in A] \qquad \frac{P \xrightarrow{\gamma} P'}{P \setminus A \xrightarrow{\gamma} P' \setminus A} \; [\gamma \notin A] \;.$$

[4] The relevant partial order is defined in Sect. 3.

These rules allow us to associate to any CSP process a labelled transition system representing its possible executions. We say that a process *diverges* if it has an infinite path whose actions are exclusively τ's. A process is *livelock-free* if it never reaches a point from which it diverges.

The denotational semantics ascribes to any CSP process a pair (T, D), where $T \subseteq \Sigma^{*\checkmark}$ is the set of visible event traces that the process may perform, and $D \subseteq T$ is the set of traces after which it may diverge.[5] Following [14], we write $\mathcal{T}^{\Downarrow}$ for the set of pairs (T, D) satisfying the following axioms (where $\frown$ denotes trace concatenation):

1. $D \subseteq T$.
2. $s \frown \langle\checkmark\rangle \in D$ implies $s \in D$.
3. $T \subseteq \Sigma^{*\checkmark}$ is non-empty and prefix-closed.
4. $s \in D \cap \Sigma^*$ and $t \in \Sigma^{*\checkmark}$ implies $s \frown t \in D$.

Axiom 4 says that the set of divergences is postfix-closed. Indeed, since we are only interested in *detecting* divergence, we treat it as catastrophic and do not attempt to record any meaningful information past a point from which a process may diverge; accordingly, our semantic model takes the view that a process may perform *any* sequence of events after divergence. Thus the only reliable behaviours of a process are those in $T - D$.

Given a process P, its denotation $[\![P]\!] = (\mathsf{traces}(P), \mathsf{divergences}(P)) \in \mathcal{T}^{\Downarrow}$ is calculated by induction on the structure of P; in other words, the model $\mathcal{T}^{\Downarrow}$ is compositional. The complete list of clauses can be found in [13, Chap. 8], and moreover the traces and divergences of a process may also be extracted from the operational semantics in straightforward fashion.

Hiding a set of events $A \subseteq \Sigma$ from a process P introduces divergence if P is capable of performing an infinite unbroken sequence of events from A. Although our model only records the finite traces of a process, the finite-branching nature of our operators entails (via König's lemma) that a process may perform an infinite trace $u \in \Sigma^{\omega}$ if and only if it can perform all finite prefixes of u.

We interpret recursive processes in the standard way by introducing a partial order $\sqsubseteq$ on $\mathcal{T}^{\Downarrow}$. We write $(T_1, D_1) \sqsubseteq (T_2, D_2)$ if $T_2 \subseteq T_1$ and $D_2 \subseteq D_1$. In other words, the order on $\mathcal{T}^{\Downarrow}$ is reverse inclusion on both the trace and the divergence components. The bottom element of $(\mathcal{T}^{\Downarrow}, \sqsubseteq)$ is $(\Sigma^{*\checkmark}, \Sigma^{*\checkmark})$, i.e., the denotation of the immediately divergent process DIV. The least upper bound of a family $\{(T_i, D_i) \mid i \in I\}$ is given by $\bigsqcup_{i \in I}(T_i, D_i) = (\bigcap_{i \in I} T_i, \bigcap_{i \in I} D_i)$.

It is readily verified that each n-ary CSP operator other than recursion can be interpreted as a Scott-continuous function $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. By induction we have that any CSP expression P in variables $X_1, \ldots, X_n$ is interpreted as a Scott-continuous map $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. Recursion is then interpreted using the least fixed point operator $\mathrm{fix} : [\mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}] \to \mathcal{T}^{\Downarrow}$. For instance $[\![\mu X \bullet X]\!]$ is the least fixed

[5] Standard models of CSP also take account of the liveness properties of a process by modelling its *refusals*, i.e., the sets of events it cannot perform after a given trace. However, this information is orthogonal to our concerns: the divergences of a process are independent of its refusals—see [13, Sect. 8.4].

point of the identity function on $\mathcal{T}^{\Downarrow}$, i.e., the immediately divergent process. Our analysis of livelock-freedom is based around an alternative treatment of fixed points in terms of metric spaces.

Definition 1. *A process P is livelock-free if* $\mathsf{divergences}(P) = \emptyset$.

In what follows, we make repeated use of standard definitions and facts concerning metric spaces. We refer the reader who might be unfamiliar with this subject matter to the accessible text [18].

Let $F(X)$ be a CSP term with a free variable X. F can be seen as a selfmap of $\mathcal{T}^{\Downarrow}$. Assume that there exists some metric on $\mathcal{T}^{\Downarrow}$ which is complete and under which F is a contraction[6]. Then it follows from the Banach fixed point theorem that F has a unique (possibly divergent) fixed point $\mu X \,.\, F(X)$ in $\mathcal{T}^{\Downarrow}$.

There may be several such metrics, or none at all. Fortunately, a class of suitable metrics can be systematically elicited from the sets of guards of a particular recursion. Roughly speaking, the metrics that we consider are all variants of the well-known 'longest common prefix' metric on traces[7], which were first studied by Roscoe in his doctoral dissertation [12], and independently by de Bakker and Zucker [1]. The reason we need to consider such variants is that hiding fails to be nonexpansive in the 'longest common prefix' metric. For instance, the distance between the traces $\langle a, a, b\rangle$ and $\langle a, a, c\rangle$ is $\frac{1}{4}$. However, after the event a is hidden, the distance becomes 1. The solution, in this particular case, is to change the definition of the length of a trace by only counting non-a events. To formalise these ideas let us introduce a few auxiliary definitions. These are all parametric in a given set of events $U \subseteq \Sigma$.

Given a trace $s \in \Sigma^{*\checkmark}$, the U-length of s, denoted $\mathsf{length}_U(s)$, is defined to be the number of occurrences of events from U occurring in s. Given a set of traces $T \subseteq \Sigma^{*\checkmark}$ and $n \in \mathbb{N}$ the restriction of T to U-length n is defined by $T \restriction_U n \mathrel{\widehat{=}} \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}$. We extend this restriction operator to act on our semantic domain $\mathcal{T}^{\Downarrow}$ by defining $(T, D) \restriction_U n \mathrel{\widehat{=}} (T', D')$, where

1. $D' = D \cup \{s^\frown t \mid s \in T \cap \Sigma^* \text{ and } \mathsf{length}_U(s) = n\}$.
2. $T' = D' \cup \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}$.

Thus $P \restriction_U n$ denotes a process which behaves like P until n events from the set U have occurred, after which it diverges. It is the least process which agrees with P on traces with U-length no greater than n.

We now define a metric d_U on $\mathcal{T}^{\Downarrow}$ by

$$d_U(P, Q) \mathrel{\widehat{=}} \inf\{2^{-n} \mid P \restriction_U n = Q \restriction_U n\} \ ,$$

where the infimum is taken in the interval $[0, 1]$.

Notice that the function $U \mapsto d_U$ is antitone: if $U \subseteq V$ then $d_U \geqslant d_V$. In particular, the greatest of all the d_U is $d_\emptyset$; this is the discrete metric on $\mathcal{T}^{\Downarrow}$.

[6] A selfmap F on a metric space $(\mathcal{T}^{\Downarrow}, d)$ is a *contraction* if there exists a non-negative constant $c < 1$ such that, for any $P, Q \in \mathcal{T}^{\Downarrow}$, $d(F(P), F(Q)) \leqslant c \cdot d(P, Q)$.

[7] In this metric the distance between two traces s and t is the infimum in $[0, 1]$ of the set $\{2^{-k} \mid s \text{ and } t \text{ possess a common prefix of length } k\}$.

Furthermore, the least of all the d_U is d_Σ; this is the standard metric on $\mathcal{T}^{\Downarrow}$ as defined in [13, Chap. 8].

Proposition 2. *Let $U \subseteq \Sigma$. Then $\mathcal{T}^{\Downarrow}$ equipped with the metric d_U is a complete ultrametric space and the set of livelock-free processes is a closed subset of $\mathcal{T}^{\Downarrow}$. Furthermore if $F : \mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}$ is contractive with respect to d_U then F has a unique fixed point given by $\lim_{n\to\infty} F^n(STOP)$. (Note that this fixed point may be divergent.)*

In the rest of this paper, the only metrics we are concerned with are those associated with some subset of Σ; accordingly, we freely identify metrics and sets when the context is unambiguous.

4 Static Livelock Analysis

We present an algorithm based on a static analysis which conservatively flags processes that may livelock. In other words, any process classified as livelock-free really is livelock-free, although the converse may not hold.

Divergent behaviours originate in three different ways, two of which are non-trivial. The first is through direct use of the process DIV; the second comes from unguarded recursions; and the third is through hiding an event, or set of events, which the process can perform infinitely often to the exclusion of all others.

Roscoe [13, Chap. 8] addresses the second and third points by requiring that all recursions be *guarded*, i.e., always perform some event prior to recursing, and by banning use of the hiding operator. Our idea is to extend Roscoe's requirement that recursions should be guarded by stipulating that one may never hide *all* the guards. In addition, one may not hide a set of events which a process is able to perform infinitely often to the exclusion of all others. This will therefore involve a certain amount of book-keeping.

We first treat the issue of guardedness of the recursions. Our task is complicated by the renaming operator, in that a purported guard may become hidden only after several unwindings of a recursion. The following example illustrates some of the ways in which a recursion may fail to be guarded, and thus diverge.

Example 3. Let $\Sigma = \{a, b, a_0, a_1, \ldots, a_n\}$ and let $R = \{(a_i, a_{i+1}) | 0 \leqslant i < n\}$ and $S = \{(a, b), (b, a)\}$ be renaming relations on Σ. Consider the following processes.

1. $\mu X \bullet X$.
2. $\mu X \bullet a \longrightarrow (X \setminus a)$.
3. $\mu X \bullet (a \longrightarrow (X \setminus b)) \sqcap (b \longrightarrow (X \setminus a))$.
4. $\mu X \bullet (a_0 \longrightarrow (X \setminus a_n)) \sqcap (a_0 \longrightarrow X[R])$.
5. $\mu X \bullet SKIP \sqcap a \longrightarrow (X \fatsemi (X[S] \setminus b))$.

The first recursion is trivially unguarded. In the second recursion the guard a is hidden after the first recursive call. In the third process the guard in each summand is hidden in the other summand; this process will also diverge once it has performed a single event. In the fourth example we cannot choose a set of

guards which is both stable under the renaming operator and does not contain a_n. This process, call it P, makes the following sequence of visible transitions:

$$P \xrightarrow{a_0} P \setminus a_n \xrightarrow{a_0} P[R] \setminus a_n \xrightarrow{a_1} P[R][R] \setminus a_n \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} P[R][R]\ldots[R] \setminus a_n.$$

But the last process diverges, since P can make an infinite sequence of a_0-transitions which get renamed to a_n by successive applications of R and are then hidden at the outermost level.

A cursory glance at the last process might suggest that it is guarded in $\{a\}$. However, similarly to the previous example, hiding and renaming conspire to produce divergent behaviour. In fact the process, call it P, can make an a-transition to $P \fatsemi (P[S] \setminus b)$, and thence to $(P[S] \setminus b)[S] \setminus b$ via two τ-transitions. But this last process can diverge.

Given a variable X and a CSP term $P = P(X)$, we aim to define inductively a collection $\mathsf{C}_X(P)$ of metrics for which P is contractive as a function of X (bearing in mind that processes may have several free variables). It turns out that it is first necessary to identify those metrics in which P is merely nonexpansive as a function of X, the collection of which we denote $\mathsf{N}_X(P)$. Intuitively, the role of $\mathsf{N}_X(P)$ is to keep track of all hiding and renaming in P. A set $U \subseteq \Sigma$ then induces a metric d_U under which P is contractive in X provided P is nonexpansive in U and $\mu X \mathbin{.} P$ always communicates an event from U prior to recursing.

The collections of metrics that we produce are conservative, i.e., sound, but not necessarily complete. As the examples above suggest, their calculation is made somewhat complicated by the possibility of recursing under renaming. For reasons that will soon become apparent, $\mathsf{N}_X(P)$ and $\mathsf{C}_X(P)$ consist of sets of *pairs* of metrics, or in other words are identified with subsets of $\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$. The key property of the function N_X is given by the following:

Proposition 4. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. If $(U, V) \in \mathsf{N}_X(P)$, then for all $T_1, T_2, \theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, $d_U(T_1, T_2) \geq d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta}))$.*

For R a renaming relation on Σ and $U \subseteq \Sigma$, let $R(U) = \{y \mid \exists x \in U \mathbin{.} x \; R \; y\}$. $\mathsf{N}_X(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ is then computed through the following inductive clauses:

$$\mathsf{N}_X(P) \mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \quad \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:}$$

$$\begin{aligned}
\mathsf{N}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{N}_X(P) \\
\mathsf{N}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{N}_X(P_1) \cap \mathsf{N}_X(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, \fatsemi, \underset{A}{\|}\} \\
\mathsf{N}_X(P \setminus A) &\mathrel{\widehat{=}} \{(U, V \cup V') \mid (U, V) \in \mathsf{N}_X(P) \wedge V \cap A = \emptyset\} \\
\mathsf{N}_X(P[R]) &\mathrel{\widehat{=}} \{(U, R(V) \cup V') \mid (U, V) \in \mathsf{N}_X(P)\} \\
\mathsf{N}_X(X) &\mathrel{\widehat{=}} \{(U, V) \mid U \subseteq V\} \\
\mathsf{N}_X(\mu Y \mathbin{.} P) &\mathrel{\widehat{=}} \{(U \cap U', V \cup V') \mid (U, V) \in \mathsf{N}_X(P) \wedge (V, V) \in \mathsf{N}_Y(P)\} \\
&\qquad \text{if } Y \neq X \ .
\end{aligned}$$

The proof of Prop. 4 proceeds by structural induction on P and can be found in the full version of the paper [11].

Before defining $\mathsf{C}_X(P)$, we need an auxiliary construct denoted $\mathsf{G}(P)$. Intuitively, $\mathsf{G}(P) \subseteq \mathcal{P}(\Sigma)$ lists the 'guards' of $\checkmark$ for P. Formally:

Proposition 5. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. If $U \in \mathsf{G}(P)$, then, with any processes—and in particular DIV—substituted for the free variables of P, P must communicate an event from U before it can do a $\checkmark$.*

The inductive clauses for G are given below. Note that these make use of the collection of *fair sets* $\mathsf{F}(P_i)$ of P_i, which is presented later on. The definition is nonetheless well-founded since F is here only applied to subterms. The salient feature of $\mathsf{F}(P_i) \neq \emptyset$ is that the process P_i is guaranteed to be livelock-free.

$$
\begin{aligned}
\mathsf{G}(STOP) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \\
\mathsf{G}(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{G}(P) \cup \{V \mid a \in V\} \\
\mathsf{G}(SKIP) &\mathrel{\widehat{=}} \emptyset \\
\mathsf{G}(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{G}(P_1) \cap \mathsf{G}(P_2) \quad \text{if } \oplus \in \{\Box, \sqcap\} \\
\mathsf{G}(P_1 \fatsemi P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if } P_1 \text{ is closed and } \mathsf{F}(P_1) \neq \emptyset \\ \mathsf{G}(P_1) & \text{otherwise} \end{cases} \\
\mathsf{G}(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if, for } i = 1, 2,\ P_i \text{ is closed and } \mathsf{F}(P_i) \neq \emptyset \\ \mathsf{G}(P_1 S) \cap \mathsf{G}(P_2) & \text{otherwise} \end{cases} \\
\mathsf{G}(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \{V \cup V' \mid V \in \mathsf{G}(P) \wedge V \cap A = \emptyset\} & \text{if } P \text{ is closed and} \\ & (\emptyset, \Sigma - A) \in \mathsf{F}(P) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{G}(P[R]) &\mathrel{\widehat{=}} \{R(V) \cup V' \mid V \in \mathsf{G}(P)\} \\
\mathsf{G}(X) &\mathrel{\widehat{=}} \emptyset \\
\mathsf{G}(\mu\, X \bullet P) &\mathrel{\widehat{=}} \mathsf{G}(P)\ .
\end{aligned}
$$

We are now ready to define $C_X(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, whose central property is given by the following proposition.

Proposition 6. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. If $(U, V) \in \mathsf{C}_X(P)$, then for all $T_1, T_2, \theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, $\frac{1}{2} d_U(T_1, T_2) \geq d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta}))$.*

$$
\begin{aligned}
\mathsf{C}_X(P) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \quad \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:} \\
\mathsf{C}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{C}_X(P) \cup \{(U, V) \in \mathsf{N}_X(P) \mid a \in V\} \\
\mathsf{C}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap \mathsf{C}_X(P_2) \quad \text{if } \oplus \in \{\Box, \sqcap, \underset{A}{\parallel}\} \\
\mathsf{C}_X(P_1 \fatsemi P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap (\mathsf{C}_X(P_2) \cup \{(U, V) \in \mathsf{N}_X(P_2) \mid V \in \mathsf{G}(P_1)\}) \\
\mathsf{C}_X(P \setminus A) &\mathrel{\widehat{=}} \{(U, V \cup V') \mid (U, V) \in \mathsf{C}_X(P) \wedge V \cap A = \emptyset\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_X(P[R]) &\mathrel{\widehat{=}} \{(U, R(V) \cup V') \mid (U,V) \in \mathsf{C}_X(P)\} \\
\mathsf{C}_X(X) &\mathrel{\widehat{=}} \emptyset \\
\mathsf{C}_X(\mu Y \mathbin{.} P) &\mathrel{\widehat{=}} \{(U \cap U', V \cup V') \mid (U,V) \in \mathsf{C}_X(P) \wedge (V,V) \in \mathsf{N}_Y(P)\} \\
&\qquad \text{if } Y \neq X \ .
\end{aligned}
$$

Note that contraction guarantees a unique fixed point, albeit not necessarily a livelock-free one. For instance, $P(X) = (a \longrightarrow X \setminus b) \,\Box\, (\mu Y \mathbin{.} b \longrightarrow Y)$ has a unique fixed point which can diverge after a single event.

In order to prevent livelock, we must ensure that, whenever a process can perform an infinite[8] unbroken sequence of events from a particular set A, then we never hide the whole of A. To this end, we now associate to each CSP term P a collection of (pairs of) *fair sets* $\mathsf{F}(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$: intuitively, this allows us to keep track of the events which the process is guaranteed to perform infinitely often in any infinite execution of P.

Given a set $W \subseteq \Sigma$, we say that a process is W-fair if any of its infinite traces contains infinitely many events from W. We now have:

Proposition 7. *Let $P(X_1, \ldots, X_n) = P(\overline{X})$ be a CSP term whose free variables are contained within the set $\{X_1, \ldots, X_n\}$. If $(U,V) \in \mathsf{F}(P)$, then, for any collection of livelock-free, U-fair processes $\theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, the process $P(\theta_1, \ldots, \theta_n)$ is livelock-free and V-fair.*

$$
\begin{aligned}
\mathsf{F}(STOP) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \\
\mathsf{F}(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{F}(P) \\
\mathsf{F}(SKIP) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \\
\mathsf{F}(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{F}(P_1) \cap \mathsf{F}(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, \fatsemi\} \\
\mathsf{F}(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} (\mathsf{F}(P_1) \cap \mathsf{F}(P_2)) \cup \\
&\qquad \{(U_1 \cap U_2, V_1) \mid (U_1, V_1) \in \mathsf{F}(P_1) \wedge (U_2, A) \in \mathsf{F}(P_2)\} \cup \\
&\qquad \{(U_1 \cap U_2, V_2) \mid (U_2, V_2) \in \mathsf{F}(P_2) \wedge (U_1, A) \in \mathsf{F}(P_1)\} \\
\mathsf{F}(P \setminus A) &\mathrel{\widehat{=}} \{(U, V \cup V') \mid (U,V) \in \mathsf{F}(P) \wedge V \cap A = \emptyset\} \\
\mathsf{F}(P[R]) &\mathrel{\widehat{=}} \{(U, R(V) \cup V') \mid (U,V) \in \mathsf{F}(P)\} \\
\mathsf{F}(X) &\mathrel{\widehat{=}} \{(U,V) \mid U \subseteq V\} \\
\mathsf{F}(\mu X \mathbin{.} P) &\mathrel{\widehat{=}} \begin{cases} \{(U \cap U', U \cup V') \mid (U,U) \in \mathsf{C}_X(P) \cap \mathsf{F}(P)\} & \text{if } \mu X \mathbin{.} P \text{ is open} \\ \mathcal{P}(\Sigma) \times \{U \cup V' \mid (U,U) \in \mathsf{C}_X(P) \cap \mathsf{F}(P)\} & \text{otherwise} \ . \end{cases}
\end{aligned}
$$

We now obtain one of our main results as an immediate corollary:

Theorem 8. *Let P be a CSP process (i.e., closed term) not containing DIV in its syntax. If $\mathsf{F}(P) \neq \emptyset$, then P is livelock-free.*

[8] Recall our understanding that a process can 'perform' an infinite trace iff it can perform all its finite prefixes.

5 Structurally Finite-State Processes

The techniques developed in Sections 3 and 4 allow us to handle the widest range of CSP processes; among others, they enable one to establish livelock-freedom of numerous infinite-state processes including examples making use of infinite buffers or unbounded counters—see [11] for examples. Such processes are of course beyond the reach of explicit-state model checkers such as FDR. In order to create them in CSP, it is necessary to use devices such as recursing under the parallel operator. In practice, however, the vast majority of processes tend to be finite state.

Let us therefore define a CSP process to be *structurally finite state* if it never syntactically recurses under any of parallel, the left-hand side of a sequential composition, hiding, or renaming.

More precisely, we first define a notion of *sequential* CSP terms: $STOP$, $SKIP$, and X are sequential; if P and Q are sequential, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \,\Box\, Q$, and $\mu X \bullet P$; and if in addition P is closed, then $P \fatsemi Q$, $P \setminus A$, and $P[R]$ are sequential. Observe that sequential processes give rise to labelled transition systems of size linear in the length of their syntax.

Now any closed sequential process is deemed to be structurally finite state; and if P and Q are structurally finite state, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \,\Box\, Q$, $P \underset{A}{\parallel} Q$, $P \fatsemi Q$, $P \setminus A$, and $P[R]$. Note that structurally finite-state CSP terms are always closed, i.e., are processes.

Whether a given process is structurally finite state can easily be established by syntactic inspection. For such processes, it turns out that we can substantially both simplify and sharpen our livelock analysis. More precisely, the computation of nonexpansive and contractive data is circumvented by instead directly examining closed sequential components in isolation. Furthermore, the absence of free variables in compound processes makes some of the earlier fairness calculations unnecessary, thereby allowing more elaborate and finer data to be computed efficiently, as we now explain.

Let u be an infinite trace over Σ, and let $F, C \subseteq \Sigma$ be two sets of events. We say that u is *fair in* F if, for each $a \in F$, u contains infinitely many occurrences of a,[9] and we say that u is *co-fair in* C if, for each $b \in C$, u contains only finitely many occurrences of b.

Given a structurally finite-state process P, we compute a collection of *fair/co-fair* pairs of disjoint sets $\Phi(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, together with a Boolean-valued *livelock flag* $\delta(P) \in \{\mathrm{true}, \mathrm{false}\}$, giving rise to our second main result:

Theorem 9. *Let P be a structurally finite-state process. Write $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\}$. If $\delta(P) = \mathrm{false}$, then P is livelock-free, and moreover, for every infinite trace u of P, there exists $1 \leq i \leq k$ such that u is both fair in F_i and co-fair in C_i.*

The calculation of $\Phi(P)$ proceeds inductively as follows. For P a closed sequential process, $\Phi(P)$ is computed directly from the labelled transition system

[9] Note that this notion of 'fairness' differs from that used in the previous section.

associated with P.[10] Otherwise:

$$\begin{aligned}
\Phi(a \longrightarrow P) &\mathrel{\widehat{=}} \Phi(P) \\
\Phi(P_1 \oplus P_2) &\mathrel{\widehat{=}} \Phi(P_1) \cup \Phi(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, \fatsemi\} \\
\Phi(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} \{(F, C) \mid F \cap C = \emptyset \wedge (F_i, C_i) \in \Phi(P_i) \text{ for } i = 1, 2 \wedge \\
&\qquad\qquad F = F_1 \cup F_2 \wedge \\
&\qquad\qquad C = (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))\} \cup \\
&\qquad \{(F, C) \mid (F, C) \in \Phi(P_1) \wedge F \cap A = \emptyset\} \cup \\
&\qquad \{(F, C) \mid (F, C) \in \Phi(P_2) \wedge F \cap A = \emptyset\} \\
\Phi(P \setminus A) &\mathrel{\widehat{=}} \{(F - A, C \cup A) \mid (F, C) \in \Phi(P)\} \\
\Phi(P[R]) &\mathrel{\widehat{=}} \{(F, C) \mid (F', C') \in \Phi(P) \wedge F' \subseteq R^{-1}(F) \wedge F \subseteq R(F') \wedge \\
&\qquad\qquad C = \{b \in \Sigma \mid R^{-1}(b) \subseteq C'\}\} \ .
\end{aligned}$$

Note that by construction, all fair/co-fair pairs of sets thus generated remain disjoint; this is key in the rule for parallel composition, where the fair/co-fair data of individual sub-components enables one to rule out certain pairs for the resulting parallel process.

The calculation of $\delta(P)$ similarly proceeds inductively, making use of the fair/co-fair data, as follows. If P is a closed sequential process, then $\delta(P)$ is determined directly from the labelled transition system associated with P, according to whether the latter contains a τ-cycle or not (using, e.g., Tarjan's algorithm). Otherwise:

$$\begin{aligned}
\delta(a \longrightarrow P) &\mathrel{\widehat{=}} \delta(P) \\
\delta(P_1 \oplus P_2) &\mathrel{\widehat{=}} \delta(P_1) \vee \delta(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, \underset{A}{\parallel}, \fatsemi\} \\
\delta(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \text{false} & \text{if } \delta(P) = \text{false and, for each } (F, C) \in \Phi(P),\ F - A \neq \emptyset \\ \text{true} & \text{otherwise}^{11} \end{cases} \\
\delta(P[R]) &\mathrel{\widehat{=}} \delta(P) \ .
\end{aligned}$$

[10] It is worth pointing out how this can be achieved efficiently. Given a set $L \subseteq \Sigma$ of events, delete all $(\Sigma - L)$-labelled transitions from P's labelled transition system. If the resulting graph contains a (not necessarily reachable) strongly connected component which comprises every single event in L, include $(L, \Sigma - L)$ as a fair/co-fair pair for P.

Of course, in actual implementations, it is not necessary to iterate explicitly over all possible subsets of Σ. The computation we described can be carried out symbolically using a Boolean circuit of size polynomial in $|\Sigma|$, using well-known circuit algorithms for computing the transitive closure of relations. Consequently, $\Phi(P)$ can be represented symbolically and compactly either as a BDD or a propositional formula. Further implementation details are provided in Sect. 6.

[11] Let us remark that the clause for the hiding operator is phrased here so as to make the rule as intuitively clear as possible. In practice, one however need not iterate

Theorems 8 and 9 yield a conservative algorithm for livelock-freedom: given a CSP process P (which we will assume does not contain DIV in its syntax), determine first whether P is structurally finite state. If so, assert that P is livelock-free if $\delta(P) = \text{false}$, and otherwise report an inconclusive result. If P is not structurally finite state, assert that P is livelock-free if $\mathsf{F}(P) \neq \emptyset$, and otherwise report an inconclusive result.

It is perhaps useful to illustrate how the inherent incompleteness of our procedure can manifest itself in very simple ways. For example, let $P = a \longrightarrow Q$ and $Q = (a \longrightarrow P) \,\Box\, (b \longrightarrow Q)$, and let $R = (P \underset{\{a,b\}}{\parallel} Q) \setminus b$. Using Bekič's procedure, R is readily seen to be (equivalent to) a structurally finite-state process. Moreover, R is clearly livelock-free, yet $\delta(R) = \text{true}$ and $\mathsf{F}(R) = \emptyset$. Intuitively, establishing livelock-freedom here requires some form of state-space exploration, to see that the 'divergent' state $(Q \underset{\{a,b\}}{\parallel} Q) \setminus b$ of R is in fact unreachable, but that is precisely the sort of reasoning that our static analysis algorithm is not geared to do.

Nonetheless, we have found in practice that our approach succeeded in establishing livelock-freedom for a wide range of existing benchmarks; we report on some of our experiments in Sect. 6.

Finally, it is worth noting that, for structurally finite-state processes, Theorem 9 is stronger than Theorem 8—it correctly classifies a larger class of processes as being livelock-free—and empirically has also been found to yield faster algorithms.

6 Implementation and Experimental Results

Computationally, the crux of our algorithm revolves around the manipulation of sets. We have built both BDD-based and propositional-formula-based implementations, using respectively CUDD 2.4.2 and MiniSat 2.0 for computations. Our resulting tool was christened SLAP, for Static Livelock Analyser of Processes.

We experimented with a wide range of benchmarks, including parameterised, parallelised, and piped versions of Milner's Scheduler, the Alternating Bit Protocol, the Sliding Window Protocol, the Dining Philosophers, Yantchev's Mad Postman Algorithm, as well as a Distributed Database algorithm.[12] In all our examples, internal communications were hidden, so that livelock-freedom can be viewed as a progress or liveness property. All benchmarks were livelock-free, although the reader familiar with the above examples will be aware that manually establishing livelock-freedom for several of these can be a subtle exercise.

over all possible pairs $(F, C) \in \Phi(P)$: it is simpler instead to evaluate the negation, an existential calculation which is easily integrated within either a SAT or BDD implementation.

[12] Scripts and descriptions for all benchmarks are available from the website associated with [14]; the reader may also wish to consult [11] for further details on our case studies.

In all cases apart from the Distributed Database algorithm, SLAP was indeed correctly able to assert livelock-freedom (save for rare instances of timing out). (Livelock-freedom for the Distributed Database algorithm turns out to be remarkably complex; see [13] for details.) In almost all instances, both BDD-based and SAT-based implementations of SLAP substantially outperformed the state-of-the-art CSP model checker FDR, often completing orders of magnitude faster. On the whole, BDD-based and SAT-based implementations performed comparably, with occasional discrepancies. All experiments were carried out on a 3.07GHz Intel Xeon processor running under Ubuntu with 8 GB of RAM. Times in seconds are given in Table 1, with * indicating a 30-minute timeout. Further details of the experiments are provided in [11].

Benchmark	**FDR**	**SLAP** (BDD)	**SLAP** (SAT)
Milner-15	0	0.19	0.16
Milner-20	409	0.63	0.34
Milner-21	948	0.73	0.22
Milner-22	*	0.89	0.25
Milner-25	*	1.63	0.55
Milner-30	*	7.34	1.14
ABP-0	0	0.03	0.03
ABP-0-inter-2	0	0.03	0.04
ABP-0-inter-3	23	0.03	0.06
ABP-0-inter-4	*	0.03	0.07
ABP-0-inter-5	*	0.03	0.08
ABP-0-pipe-2	0	0.03	0.08
ABP-0-pipe-3	2	0.04	0.12
ABP-0-pipe-4	175	0.04	0.23
ABP-0-pipe-5	*	0.05	0.34
ABP-4	0	0.11	0.92
ABP-4-inter-2	39	0.12	1.49
ABP-4-inter-3	*	0.13	1.71
ABP-4-inter-7	*	0.15	3.68
ABP-4-pipe-2	12	0.13	2.96
ABP-4-pipe-3	*	0.15	6.34
ABP-4-pipe-7	*	0.25	31.5

Benchmark	**FDR**	**SLAP** (BDD)	**SLAP** (SAT)
SWP-1	0	0.03	0.08
SWP-2	0	0.34	*
SWP-3	0	40.94	*
SWP-1-inter-2	0	0.04	0.12
SWP-1-inter-3	31	0.04	0.16
SWP-1-inter-4	*	0.05	0.19
SWP-1-inter-7	*	0.06	0.33
SWP-2-inter-2	170	0.47	*
SWP-2-inter-3	*	0.64	*
SWP-1-pipe-3	0	0.04	0.47
SWP-1-pipe-4	3	0.05	0.73
SWP-1-pipe-5	246	0.05	1.10
SWP-1-pipe-7	*	0.06	2.89
Philosophers-7	2	1.64	0.20
Philosophers-8	20	2.46	0.31
Philosophers-9	140	3.99	0.46
Philosophers-10	960	7.39	0.61
Mad Postman-2	0	0.06	0.04
Mad Postman-3	6	*	0.23
Mad Postman-4	*	*	1.11
Mad Postman-5	*	*	5.67
Mad Postman-6	*	*	27.3

Table 1. Times reported are in seconds, with * denoting a 30-minute timeout.

7 Future Work

A interesting property of our approach is the possibility for our algorithm to produce a *certificate* of livelock-freedom, consisting among others in the various

sets supporting the final judgement. Such a certificate could then be checked in polynomial time by an independent tool.

Other directions for future work include improving the efficiency of SLAP by incorporating various abstractions (such as collapsing all events on a given channel, or placing *a priori* bounds on the size of sets), or conversely increasing accuracy at modest computational cost, for example by making use of algebraic laws at the syntactic level, such as bounded unfoldings of parallel compositions.

References

[1] J. W. De Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[2] M. Bravetti and R. Gorrieri. Deciding and axiomatizing weak ST bisimulation for a process algebra with recursion and action refinement. *ACM Transactions on Computational Logic*, 3(4):465–520, 2002.

[3] A. Dimovski. A compositional method for deciding program termination. In *ICT Innovations*, volume 83, pages 71–80. Springer CCIS, 2010.

[4] R. O. Gandy. An early proof of normalization by A.M. Turing. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, volume 267, pages 453–455. Academic Press, 1980.

[5] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Science 7. Cambridge University Press, 1988.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.

[7] S. Leue, A. Ştefănescu, and W. Wei. A livelock freedom analysis for infinite state asynchronous reactive systems. In *Proceedings of CONCUR*, volume 4137, pages 79–94. Springer LNCS, 2006.

[8] S. Leue, A. Ştefănescu, and W. Wei. Dependency analysis for control flow cycles in reactive communicating processes. In *Proceedings of SPIN*, volume 5156. Springer LNCS, 2008.

[9] R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[10] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996.

[11] J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. A static analysis framework for livelock freedom in CSP. *Logical Methods in Computer Science*, ???(???), 2013.

[12] A. W. Roscoe. *A Mathematical Theory of Communicating Processes*. PhD thesis, Oxford University, 1982.

[13] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, London, 1997.

[14] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2011. `www.cs.ox.ac.uk/ucs/`.

[15] D. Sangiorgi. Types, or: Where's the difference between CCS and π? In *Proceedings of CONCUR 02*, volume 2421, pages 76–97. Springer LNCS, 2002.

[16] S. Schneider, H. Treharne, and H. Wehrheim. A CSP approach to control in Event-B. In *Proceedings of IFM*, volume 6396. Springer CCIS, 2010.

[17] S. Schneider, H. Treharne, and H. Wehrheim. A CSP account of Event-B refinement. Unpublished, 2011.

[18] W. A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 1975.
[19] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
[20] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the π-Calculus. In *Proceedings of LICS 01*, pages 311–322. IEEE Computer Society Press, 2001.