# Efficient Deadlock Checking using Local Analysis and SAT Solving — Technical Report

Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe

Department of Computer Science, University of Oxford, UK
{pedro.antonino,thomas.gibson-robinson,bill.roscoe}@cs.ox.ac.uk

**Abstract.** We build upon established techniques of deadlock analysis by formulating a new sound but incomplete framework for deadlock freedom analysis that tackles some sources of imprecision of current incomplete techniques. Our new deadlock candidate criterion is based on constraints derived from the analysis of the state space of pairs of components. This new characterisation represents an improvement in the accuracy of current incomplete techniques; in particular, non-hereditary deadlock-free systems, which are neglected by most incomplete techniques, are tackled by our framework. Furthermore, we demonstrate how SAT checkers can be used to efficiently implement our framework in a way that, typically, scales better than current techniques for deadlock analysis. This is demonstrated by a series of practical experiments.

## 1 Introduction

Deadlock freedom usually corresponds to a first step towards analysing the correctness of a concurrent system. A system is deadlock free if and only if it cannot reach a state in which it is stuck. Moreover, many safety properties can be reduced to verifying deadlock freedom of modified systems [11]. Unsurprisingly, even when restricted to deadlock analysis, existing automated verification techniques still suffer from the state explosion problem.

Incomplete techniques for deadlock analysis [5, 14, 13] have been proposed in attempts to circumvent the state explosion problem. These frequently scale far better than the full state analysis required by model checking, and are sound in proving deadlock freedom, but (i) tend not to provide examples of deadlocks when they fail and (ii) can fail even for some deadlock-free systems: the latter is what is meant by "incomplete". One can see this incompleteness as the price to pay for achieving scalability.

Current incomplete techniques are built around the principle that a deadlock state, under reasonable assumptions, always presents a cycle of *ungranted requests* between components of the system[1]. An ungranted request arises from a component to another if and only if the former is trying to communicate with the

[1] Depending on the properties of the underlying communicating system, one might be able to restrict such cycles to *proper cycles* which have at least three nodes, and where all the nodes are distinct.

latter, but they cannot agree on any event. To prove the absence of such a cycle, these methods rely on local properties of the system, derived from the analysis of individual components or pairs of them, to (either explicitly or implicitly) construct and analyse a dependency graph. These approaches have two important sources of imprecision. Firstly, under our assumptions, a cycle is a necessary condition for a deadlock state but not a sufficient one. So, despite being deadlock free, some deadlock-free systems are bound to present these cycles and, as such, they cannot be handled by these methods. For instance, non-hereditary deadlock free systems, namely, deadlock-free systems whose subsystems might deadlock, cannot be tackled by current techniques using local analysis. Secondly, to keep the analysis of these dependency graphs efficient, some local properties that could be used to improved accuracy are ignored because they focus on proposing polynomially checkable conditions in terms of the local information collected.

In this paper, we present a new incomplete method for establishing deadlock freedom that alleviates the mentioned sources of imprecision of current techniques. Instead of looking for cycles, we look for *complete snapshots* of the system that are fully consistent with derived local properties. A complete snapshot is an assignment of component states to components that depicts a possible state of the concurrent system. Unlike others, our method uses a condition that is not known to be polynomially checkable. While unsurprising in itself, this new criterion has proved to be efficiently determinable using the power of SAT checking. Our work has been inspired by Martin's definition of the State Dependency Digraph [14] (see Section 3), and by the successful use of SAT checkers for livelock analysis reported in [16].

*Outline.* Section 2 briefly introduces CSP's operational semantics, which is the formalism upon which our strategy is based. However, this paper can be understood purely in terms of communicating systems of LTSs, and knowledge of CSP is not a prerequisite. Section 3 presents some current incomplete techniques for deadlock analysis. In Section 4, we introduce our technique. Section 5 outlines the complexity of our method, whereas Section 6 outlines the accuracy of our method. In the following section, we give an encoding of our deadlock-freedom analysis as a SAT problem. Section 8 presents some experiments conducted to assess the accuracy and efficiency of our framework. Finally, in Section 9, we present our concluding remarks.

## 2 Background

Communicating Sequential Processes (CSP) [12, 19] is a notation used to model concurrent systems where processes interact, exchanging messages. Here we describe some structures used by FDR in implementing CSP's operational semantics. As this paper does not depend on the details of CSP, we do not describe the details of the language or its semantics. These can be found in [19].

CSP's operational semantics interpret language terms in a *labelled transition system* (LTS)[2].

**Definition 1.** *A labelled transition system is a 4-tuple* $(S, \Sigma, \Delta, \hat{s})$ *where:*

- $S$ *is a set of states;*
- $\Sigma$ *is the alphabet (i.e. a set of events);*
- $\Delta \subseteq S \times \Sigma \times S$ *is a transition relation;*
- $\hat{s} \in S$ *is the starting state.*

For the purposes of this paper, the events $\tau$ (the silent event) and $\checkmark$ (the termination signal) are considered members of $\Sigma$, since there is no difference between them and regular events for the purpose of deadlock analysis, and their behaviour can be accommodated in the supercombinator framework we use.

As a convention, $\Sigma^{-} \mathrel{\widehat{=}} \Sigma \cup \{-\}$, where $- \notin \Sigma$. We write $s \xrightarrow{e} s'$ if $(s, e, s') \in \Delta$. There is a path from $s$ to $s'$ with the sequence of events $\langle e_1, \ldots, e_n \rangle$, represented by $s \xrightarrow{\langle e_1, \ldots, e_n \rangle} s'$, if there exist $s_1, \ldots, s_{n-1}$ such that $s \xrightarrow{e_1} s_1 \ldots s_{n-1} \xrightarrow{e_n} s'$. A *trace* of a transition system is a path such that the initial state is $\hat{s}$.

While CSP, in common with many other languages, can have its operational semantics given in SOS style, FDR represents them as combinators, a notation which is itself compositional and allows complex CSP constructs, including communicating systems, to be represented as *supercombinator machines.* A supercombinator machine consists of a set of component LTSs along with a set of rules that describe how the transitions should be combined. A rule combines transitions of (a subset of) the components and determines the event the machine performs. We also use these machines to analyse the behaviour of communicating systems. For simplicity in our analysis, we restrict FDR's normal definition of supercombinator machines in a way that corresponds to there being a static communicating system with all communication between components being pairwise:

**Definition 2.** *A triple-disjoint supercombinator machine is a pair* $(\mathcal{L}, \mathcal{R})$ *where:*

- $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ *is a sequence of component LTSs;*
- $\mathcal{R}$ *is a set of rules of the form* $(e, a)$ *where:*
  - $e \in (\Sigma^{-})^n$ *specifies the event that each component must perform, where* $-$ *indicates that the component performs no event.* $e$ *must also be triple-disjoint, that is, at most two components must be involved in a rule.*
    - $triple\_disjoint(e) \mathrel{\widehat{=}} \forall i, j, k : \{1 \ldots n\} \mid i \neq j \wedge j \neq k \wedge i \neq k \bullet e_i = - \vee e_j = - \vee e_k = -$
  - $a \in \Sigma$ *is the event the supercombinator performs.*

This restriction is similar to those adopted in related work to ours [14, 5]. Henceforth, we omit the mention of triple-disjoint.

Given a supercombinator machine, a corresponding LTS can be constructed.

[2] FDR uses a more general representation of a process called a *generalised labelled transition system* (GLTS). Nevertheless, this extension can be simply converted into a traditional LTS and working with LTS makes our definitions considerably simpler.

**Definition 3.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. The LTS induced by $\mathcal{S}$ is the tuple $(S, \Sigma, \Delta, \hat{s})$ such that:*

- $S = S_1 \times \ldots \times S_n$;
- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$;
- $\Delta = \{((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \mid \exists ((e_1, \ldots, e_n), a) : \mathcal{R} \bullet \forall i : \{1 \ldots n\} \bullet;$
  $(e_i = - \wedge s_i = s'_i) \vee (e_i \neq - \wedge (s_i, e_i, s'_i) \in \Delta_i)\}$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$.

From now on, we use *system state* (*component state*) to designate a state in the system's (component's) LTS.

**Definition 4.** *A LTS $(S, \Sigma, \Delta, \hat{s})$ deadlocks in a state $s$ if and only if $deadlocked(s)$ holds, where:*

- $deadlocked(s) \mathrel{\widehat{=}} reachable(s) \wedge blocked(s)$
- $reachable(s) \mathrel{\widehat{=}} \exists\, tr : \Sigma^* \bullet \hat{s} \xrightarrow{tr} s$
- $blocked(s) \mathrel{\widehat{=}} \neg \exists\, s' : S\, ;\, e : \Sigma \bullet s \xrightarrow{e} s'$

When considering the deadlock detection problem, for the sake of decidability, we only analyse supercombinator machines with a finite number of components, which are themselves represented by finite LTSs, and a finite number of rules.

## 3 Related Work

The authors of this paper have investigated the role played by local analysis in establishing deadlock freedom in [17, 7, 3, 2]. These works introduce a formalisation of design patterns that can be used for designing deadlock-free systems. Despite being efficient, as these techniques analyse components in isolation, they can be restrictive since only a handful of behavioural patterns are available.

In [5, 4, 13, 14], fully-automated incomplete techniques for deadlock freedom are introduced. These techniques are proposed for different contexts and types of concurrency: [5] proposes a method for analysing syntactically-restricted shared-variable concurrent programs, [4] adapts [5] to a more general setting meant to describe component-based message-passing systems, [13] proposes a method for architecturally-restricted component-based systems interacting via message passing, and [14] proposes a method for syntactically-restricted message-passing concurrent systems. All these methods were designed, to some extent, around the principle that, under reasonable assumptions about the system, any deadlock state would contain a proper cycle of ungranted requests. So, to prove deadlock freedom, they would use local properties of the system, derived from analysing individual components and communicating pairs of components, to construct an ungranted-requests graph and show that such a cycle cannot arise in any conceivable state of the system.

To discuss in more detail how such approaches work, we present the *SDD framework*[3] developed by Martin in [14]. We regard our framework as a development on the SDD. Martin's analysis of SDDs is one of the most general previous approaches to locally based deadlock analysis.

In that work, the local properties used to prove deadlock freedom are derived from the analysis of pairs of components, or rather a projection of the system over a pair of its components.

**Definition 5.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. The* pairwise projection *$\mathcal{S}_{i,j}$ of the machine $\mathcal{S}$ on components $i$ and $j$ is given by:*

$$\mathcal{S}_{i,j} = (\langle L_i, L_j \rangle, \{((e_i, e_j), a) | (e, a) \in \mathcal{R} \wedge (e_i \neq - \vee e_j \neq -)\})$$

In Martin's approach, a dependency digraph is constructed and then analysed for absence of cycles. The dependency digraph constructed has a node for each state of each component, and an edge from a state $s$ of component $i$ to a state $s'$ of component $j$ if and only if $reachable_{i,j}((s, s'))$ and $ungranted\_request_{i,j}(s, s')$ hold. $reachable_{i,j}$ denotes the *reachable* predicate for the LTS induced by $\mathcal{S}_{i,j}$. $ungranted\_request_{i,j}(s, s')$ holds when, in their respective states ($i$ in $s$ and $j$ in $s'$), component $i$ is willing to synchronise with $j$ (according to $\mathcal{S}_{i,j}$), but they cannot agree on any event.

Under the assumption that components neither terminate nor deadlock, a cycle of ungranted requests is a necessary condition for a system deadlock. Hence, the absence of cycles in the dependency digraph is a proof of deadlock freedom, whereas a cycle represents a potential deadlock which we call a *SDD candidate*.

**Definition 6.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. We assume $\mathcal{U}$ is the disjoint union of all $S_i$ and $s_{i,j}$ denotes the state $j$ of the component $i$. A sequence of component states $c \in \mathcal{U}^*$ is a SDD candidate if and only if for all $i \in \{1 \ldots |c|\}$, given that $c_i = s_{j,k}$ and $c_{i \oplus 1} = s_{l,m}$, $reachable_{j,l}((s_{j,k}, s_{l,m}))$ and $ungranted\_request_{j,l}(s_{j,k}, s_{l,m})$ hold, where $\oplus$ denotes addition modulo the size of $c$.*

This method can carry out deadlock-freedom verifications very efficiently: a digraph can be shown to have no cycles in linear time using a modified *depth-first-search*. This efficiency, however, comes with a price as the use of a cycle as a candidate makes this method imprecise in several ways. Firstly, a cycle might not be consistent with basic sanity conditions such as it must have a single node per component, after all no component can be in two different states in a single deadlock. Secondly, a cycle is only partially consistent with the local reachability and local blocking properties derived from the analysis of pairs of components. Note that only adjacent elements in the cycle are guaranteed to be pairwise reachable and pairwise blocked. So, there may be local properties of non-adjacent component states not tested for that might eliminate some SDD candidate. Finally, a cycle, as a necessary condition, is bound to arise in some deadlock-free systems. Thus, in such cases, this framework is ineffective. The

[3] SDD stands for State Dependency Digraph.

reason why these sources of imprecision are not addressed is that these methods look for polynomially checkable conditions for guaranteeing deadlock freedom and tackling any of these sources of imprecision is likely to make the problem of finding a candidate in the dependency digraph NP-complete.

## 4 A New Framework for Deadlock-freedom Verification using Local Analysis

In this section, we propose a new framework that is meant to address the mentioned sources of imprecision in current techniques. To achieve that, we propose a new way of detecting potential deadlocks. Instead of looking for cycles, we look for complete snapshots of the system that are fully consistent with the local reachability and blocking information. A complete snapshot is a tuple containing a component state per component in the system. So, a deadlock candidate for this new framework, which call a *pair candidate*, is given as follows.

**Definition 7.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. A state $s = (s_1, \ldots, s_n) \in S$ is a* pair candidate *if and only if $pair\_candidate(s)$ holds, where:*

- $pair\_candidate(s) \mathrel{\widehat{=}} pairwise\_reachable(s) \wedge blocked(s)$
- $pairwise\_reachable(s) \mathrel{\widehat{=}} \forall\, i, j : \{1 \ldots n\} \mid i \neq j \bullet reachable_{i,j}((s_i, s_j))$

This new characterisation creates a framework that uses more information to disprove potential deadlock candidates if compared to prior techniques using pairwise analysis of components. By analysing complete snapshots, only complete states of the system are examined, and as a consequence, our framework is able to prove that systems possessing ungranted-requests cycles are deadlock free.

Two remarks about the blocked condition deserve mention. Firstly, the blocking condition seems to be global, but in fact, it can be validated using individual and pairwise component analyses. As systems are triple disjoint, a state is blocked if and only if all components neither perform an individual event or communicate with another component. Secondly, this blocking condition is exact, so in our framework, false negatives can only arise from the fact that the derived local reachability properties cannot prove the unreachability of a candidate.

Our framework is sound, as demonstrated next.

**Lemma 1.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and we use the subscript $i, j$ to denote the elements of the LTS induced by $S_{i,j}$.*

$$\forall\, s : S\, ; tr : \Sigma^* \bullet \hat{s} \xrightarrow{tr} s \Longrightarrow pairwise\_reachable(s)$$

*Proof.* We prove by induction on the size of the path $\hat{s} \xrightarrow{tr} s$ that whenever $\hat{s} \xrightarrow{tr} s$ holds then, for any $k$ and $l$ in $\mathcal{S}$, $reachable_{k,l}(s_{k,l})$ also holds. We use $PC$ as a shorthand for predicate calculus.

– Base case: trivially, $\hat{s} \xrightarrow{\langle\rangle} \hat{s}$ and $\hat{s}_{k,l} \xrightarrow{\langle\rangle}_{k,l} \hat{s}_{k,l}$.
– Inductive case: We show that

$$\forall\, s : S\,;\, tr : \Sigma^*\,;\, a : \Sigma \bullet \hat{s} \xrightarrow{tr \frown \langle a\rangle} s \Longrightarrow \mathit{pairwise\_reachable}(s)$$

using the following inductive hypothesis

$$IH \mathrel{\widehat{=}} \forall\, s : S\,;\, tr : \Sigma^* \bullet \hat{s} \xrightarrow{tr} s \Longrightarrow \mathit{pairwise\_reachable}(s).$$

$$\begin{aligned}
&\hat{s} \xrightarrow{tr \frown \langle a\rangle} s\\
&\Longleftrightarrow \exists\, s' : S \bullet \hat{s} \xrightarrow{tr} s' \wedge s' \xrightarrow{a} s && \text{Path def}\\
&\Longrightarrow \exists\, s' : S \bullet \mathit{pairwise\_reachable}(s') \wedge s' \xrightarrow{a} s && \text{IH}\\
&\Longrightarrow \exists\, s' : S \bullet \mathit{reachable}_{k,l}(s'_{k,l}) \wedge s' \xrightarrow{a} s && \text{p.\_reach. def and PC}\\
&\Longrightarrow \exists\, s' : S \bullet \mathit{reachable}_{k,l}(s'_{k,l}) \wedge\\
&\quad (\exists((e_1,\ldots,e_n),a) : \mathcal{R} \bullet \forall\, i : \{1,\ldots,n\} \bullet\\
&\qquad (e_i = - \wedge s'_i = s_i) \vee (e_i \neq - \wedge (s'_i, e_i, s_i) \in \Delta_i)) && \Delta \text{ def}
\end{aligned}$$

At this point, there are four cases to consider for the rule $((e_1,\ldots,e_n),a)$.

- Case 1 ($e_k = - \wedge e_l = -$). From $\Delta$ definition, as components $k$ and $l$ do not participate in the transition, $s'_{(k,l)} = s_{(k,l)}$.

$$\begin{aligned}
&\Longrightarrow \exists\, s' : S \bullet \mathit{reachable}_{k,l}(s'_{k,l}) \wedge s'_{k,l} = s_{k,l}\\
&\Longrightarrow \mathit{reachable}_{k,l}(s_{k,l}) && \mathit{PC}
\end{aligned}$$

- Case 2 ($e_k = - \wedge e_l \neq -$). Based on $\Delta$ definition, we know that $s'_l \xrightarrow{e_l} s_l$ and $s'_k = s_k$. From $\mathcal{S}_{k,l}$ definition, we know that $((-,e_l),a) \in \mathcal{R}_{k,l}$. Thus, $s'_{k,l} \xrightarrow{a}_{k,l} s_{k,l}$.

$$\begin{aligned}
&\Longrightarrow \exists\, s' : S \bullet \mathit{reachable}_{k,l}(s'_{k,l}) \wedge s'_{k,l} \xrightarrow{a}_{k,l} s_{k,l}\\
&\Longrightarrow \mathit{reachable}_{k,l}(s_{k,l}) && \mathit{reachable}_{k,l} \text{ def and PC}
\end{aligned}$$

- Case 3 ($e_k \neq - \wedge e_l = -$) is symmetric to Case 2.
- Case 4. ($e_k \neq - \wedge e_l \neq -$). Based on $\Delta$ definition, we know that $s'_l \xrightarrow{e_l} s_l$ and $s'_l \xrightarrow{e_l} s_l$. From $\mathcal{S}_{k,l}$ definition, we know that $((e_k,e_l),a) \in \mathcal{R}_{k,l}$. Thus, $s'_{k,l} \xrightarrow{a}_{k,l} s_{k,l}$.

$$\begin{aligned}
&\Longrightarrow \exists\, s' : S \bullet \mathit{reachable}_{k,l}(s'_{k,l}) \wedge s'_{k,l} \xrightarrow{a}_{k,l} s_{k,l}\\
&\Longrightarrow \mathit{reachable}_{k,l}(s_{k,l}) && \mathit{reachable}_{k,l} \text{ def and PC}
\end{aligned}$$

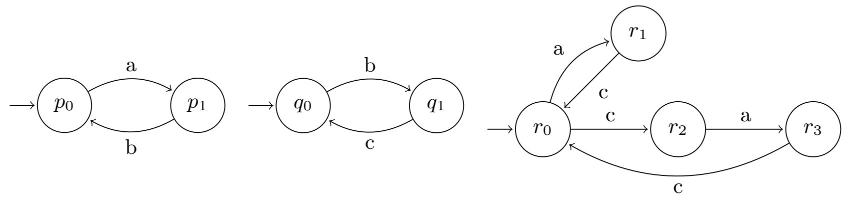

**Fig. 1.** LTSs of components $L_1$, $L_2$ and $L_3$, respectively.

□

**Theorem 1.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For any $s \in S$,*

$$\neg pair\_candidate(s) \implies \neg deadlocked(s)$$

*Proof.* We know, from *pair_candidate*'s definition, that $\neg pair\_candidate(s)$ if and only if $\neg blocked(s)$ or $\neg pairwise\_reachable(s)$. In case $\neg blocked(s)$, the claim follows easily. In case $\neg pairwise\_reachable(s)$, the claim follows thanks to the fact that for any $s \in S$, $reachable(s) \implies pairwise\_reachable(s)$, which can be, in turn, derived from Lemma 1. □

This criterion will be shown to be more accurate than the SDD one, but remains incomplete because it relies on local analysis to approximate reachability: there may well be pair candidates that are not actually reachable.

*Example 1.* Let $\mathcal{S} = (\langle L_1, L_2, L_3\rangle, \mathcal{R})$ be the supercombinator machine such that the components are described graphically in Figure 1 and they must synchronise on shared events. That is, $\mathcal{R} = \{((a, -, a), a), ((b, b, -), b), ((-, c, c), c)\}$.

For this system, the state $(s_0, s_0, s_3)$ is pairwise-reachable and blocked, but not reachable. Thus, it constitutes a pair candidate but not a deadlock. □

What we have done here is to use a characterisation of what a deadlock state looks like in conjunction with an approximation to the reachability criterion for states. What it searches for are not *reachable* deadlocks, but rather *pair*-consistent deadlocks. Therefore, we call it *Pair*. One could easily imagine using different local groups of components to determining consistency, or applying similar approaches to analyse communicating systems for individual states that have properties other than being deadlocked.

## 5 Complexity of Pair Framework

In this section, we analyse the complexity of detecting a pair candidate, given a system represented by a supercombinator machine as an input. In order to reason about our problem's complexity, we define the size of such a machine.

**Definition 8.** *The size of a supercombinator machine $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ is given by $|S| = n \cdot |L_{Max}| + |\mathcal{R}|$, where:*

– *$|L_{Max}|$ gives the size of the largest component LTS, and a LTS $(S, \Sigma, \Delta, \hat{s})$ size is given by $|S| + |\Delta|$.*

The main result of this section consists of showing that our detection problem is NP-complete, as presented next.

**Theorem 2.** *Let $\mathcal{S}$ be a supercombinator machine, and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. The problem of deciding*

$$\exists\, s : S \bullet \mathit{pair\_candidate}(s)$$

*is NP-Complete.*

*Proof.* To prove NP-completeness, we show that:

**(a)** this problem is in NP; given a state $s' \in S$, $\mathit{pair\_candidate}(s')$ validity can be verified in polynomial time on the size of the supercombinator;
**(b)** this problem is NP-hard; we prove this by showing that the CNF-satisfiability problem can be polynomially reduced to our pairwise-deadlock detection problem.

**(a)** We show that $\mathit{pair\_candidate}(s')$ can be verified in $\mathcal{O}(n^2 \cdot |L_{MAX}|^2 \cdot |R|)$ steps with the function IsPairCandidate in Algorithm 1. To this end, we present an analysis of the number of steps needed to carry out the two auxiliary functions: IsPairwiseReachable and IsBlocked.

1. IsPairwiseReachable *takes $\mathcal{O}(n^2 \cdot |L_{MAX}|^2 \cdot |R|)$ steps*;

   In this function, most of the constructs are self-explanatory and simple to analyse. However, this is not the case for establishing $\mathit{reachable}_{i,j}(s_{i,j})$ on Line 6. For carrying out this checking, one can devise a function with three steps: the creation of the pairwise projection $\mathcal{S}_{i,j}$, the construction of its induced LTS, and this LTS's state space search for $s_{i,j}$. In what follows, we analyse the complexity of these three steps.
   – The creation of $\mathcal{S}_{i,j}$ takes $\mathcal{O}(|L_{Max}| + |R|)$ steps. The copy of the pair of components should take $\mathcal{O}(|L_{Max}|)$ steps, as the copy of each component should take at most $|L_{MAX}|$ steps. The rules projection, which consists of possibly the projection of each rule, should take $\mathcal{O}(|R|)$ steps. We consider that an application or projection of a rule takes constant time due to triple-disjointness. Triple-disjointness implies that at most two component are involved, and thus we can concisely represent the event-tuple of a rule using, at most, a pair of event-components.
   – The construction of the induced LTS for $\mathcal{S}_{i,j}$ takes $\mathcal{O}(|L_{MAX}|^2 \cdot |R|)$ steps. It can be constructed by first enumerating all the states and then creating the edges by attempting to apply the rules to each of the enumerated states. The enumeration of the states should take at most $\mathcal{O}(|L_{MAX}|^2)$ steps, whereas the construction of the edges should take at most $|L_{MAX}|^2 \cdot 2 \cdot |R|$.

  - The search phase consists of a standard *breath-first-search* algorithm to find the state $s_{i,j}$, which takes $\mathcal{O}(|L_{MAX}|^2 \cdot |R|)$ steps. This search is carried out in a linear number of steps on the size of the induced LTS, and from our analysis of the construction process, this size should be bound by $|L_{MAX}|^2 + |L_{MAX}|^2 \cdot 2 \cdot |R|$: there might be at most $|L_{MAX}|^2$ states and at most $|L_{MAX}|^2 \cdot 2 \cdot |R|$ edges.

   From these analyses, we can conclude that establishing $reachable_{i,j}(s_{i,j})$ should take $\mathcal{O}(|L_{MAX}|^2 \cdot |R|)$ steps. Therefore, we can conclude that the complexity of executing the loop body in IsPairwiseReachable is $\mathcal{O}(|L_{MAX}|^2 \cdot |R|)$. As it can be repeated for at most $n^2$ times, thanks to the fact that there is only $n^2$ as many pairs of components, we conclude that the complexity of IsPairwiseReachable is given by $\mathcal{O}(n^2 \cdot |L_{MAX}|^2 \cdot |R|)$.

2. IsBlocked *takes* $\mathcal{O}(|R|)$ *steps.*

   In this function, the only construct that requires some explanation is $matches(s, r)$. This predicate holds if the rule $r$ can be applied to the state $s$. As we discussed, rule application takes constant time due to the triple-disjointness requirement. From this consideration, we can conclude that the body of the *for*-loop in IsBlocked takes $\mathcal{O}(1)$ steps to finish. Hence, as the loop can be repeated at most $|R|$ times, IsBlocked takes $\mathcal{O}(|R|)$ steps to complete.

**Algorithm 1** pair candidate verification algorithm, where $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$

```
 1: function IsPairCandidate($s : S$)
 2:     return IsPairwiseReachable(s) $\wedge$ IsBlocked(s)
 3: end function

 4: function IsPairwiseReachable($s : S$)
 5:     for all $i, j \in \{1 \ldots n\}$ such that $i \neq j$ do
 6:         if $\neg reachable_{i,j}(s_{i,j})$ then
 7:             return false
 8:         end if
 9:     end for
10:     return true
11: end function

12: function IsBlocked($s : S$)
13:     for all $r \in R$ do
14:         if $matches(s, r)$ then
15:             return false
16:         end if
17:     end for
18:     return true
19: end function
```

**(b)** In this second part of the proof, we demonstrate that our problem is NP-hard by presenting a polynomial reduction from the CNF-SAT problem to our pair candidate detection problem. To begin with, we introduce the CNF-SAT problem and some useful notation.

**Definition 9.** *Given a boolean function $\mathcal{F}(x_1, \ldots, x_m)$ with $m$ boolean variables, whose formula is in Conjunctive Normal Form, the CNF-SAT problem consists of finding an assignment to the $m$ boolean variables so that $\mathcal{F}$ holds. That is,*

$$\exists\, x_1, \ldots, x_n : \mathbb{B} \bullet \mathcal{F}(x_1, \ldots, x_m)$$

We use $\mathcal{F}$ without arguments to denote the formula associated with the function, and we manipulate this formula using the following conventions. $\mathcal{F}_i$ denotes $\mathcal{F}$'s $i$-th clause (i.e. disjunction) and $\mathcal{F}_{i,j}$ denotes its $j$-th literal in the $i$-th clause. Moreover, $|\mathcal{F}|$ denotes the size of the formula (namely, the overall number of literals in the formula), $|\mathcal{F}\_|$ denotes the number of clauses in the formula and $|\mathcal{F}_i|$ denotes the number of literals in the $i$-th clause of the formula. Finally, we use $var(i, j)$ to denote the index of the boolean variable in the literal $\mathcal{F}_{i,j}$. For instance, if $\mathcal{F}_{i,j} = \neg x_l$ then $var(i, j) = l$.

At the core of our reduction strategy is a mapping that translates, in polynomial time on the size of the formula, a boolean function into a supercombinator such that the boolean function is satisfiable if and only if the supercombinator has a Pair deadlock-candidate state. Hence, our reduction consists of deciding whether a CNF boolean function is satisfiable by translating it into a supercombinator and looking for a pair candidate state.

Our translation creates a supercombinator with a component $T(\mathcal{F}_i)$ for each clause $\mathcal{F}_i$ and a component $T(x_i)$ for each variable $x_i$ of a formula $\mathcal{F}$. The component $T(\mathcal{F}_i)$ captures the satisfiability of $\mathcal{F}_i$, whereas the component $T(x_i)$ model the assignment of $x_i$ to a boolean value. In Figure 5, we introduce two diagrams that informally define these two components. (We formally define them in Definition 10). For these diagrams, we assume that $\mathcal{F}_i = \mathcal{F}_{i,1} \vee \ldots \vee \mathcal{F}_{i,|\mathcal{F}_i|}$ is a clause of the formula $\mathcal{F}$, that $x_i$ is a variable of $\mathcal{F}$, and that $ev$ is a function from a literal to an event such that $ev(\neg x_i) = false_i$ and $ev(x_i) = true_i$.

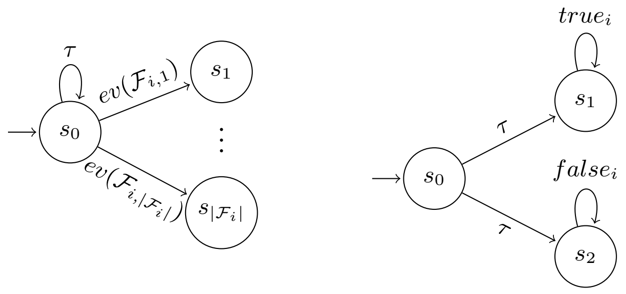

**Fig. 2.** LTSs $T(\mathcal{F}_i)$ and $T(x_i)$ respectively

Component $T(\mathcal{F}_i)$ is initially in state $s_0$, which captures that no literal of $\mathcal{F}_i$ has been satisfied, and it can transition to a final state $s_j$, which captures the fact that literal $\mathcal{F}_{i,j}$ has been satisfied, upon performing the event $ev(\mathcal{F}_{i,j})$, which denotes that the variable in the corresponding literal has been assigned to a value satisfying the literal. Component $T(x_i)$ is initially in state $s_0$, which denotes that no assignment has been made to variable $x_i$, and it can transition to either state $s_1$ or $s_2$. $s_1$ denotes that $x_i$ has been assigned to $true$, whereas $s_2$ corresponds to the assignment of $x_i$ to $false$. When in $s_1$ ($s_2$), the component can perform the event $true_i$ ($false_i$), which correspond to the assignment that has been made. So, in the translated supercombinator, the system can be seen as a set of components that capture the satisfiability of clauses and a set of components that capture assignments for boolean variables.

The set of rules provides the appropriate connection between these two sets of components. They enable the synchronisation between variable components and clauses components so that the satisfiability of the clauses is guided by the assignment of variables. The formal translation function is provided next. In the components sequence of our translated supercombinator, we use as a convention that the components in the range 1 to $|\mathcal{F}\_|$ are clause components, whereas components in the range $|\mathcal{F}\_|+1$ to $|\mathcal{F}\_|+m$ are variable ones.

**Definition 10.** *Given a boolean function $\mathcal{F}$ and its variables $x_1, \ldots, x_m$, we propose the following transformation function $Translate(\mathcal{F}, (x_1, \ldots, x_m)) = (\mathcal{L}, \mathcal{R})$, where:*

- $\mathcal{L} = \langle T(\mathcal{F}_1), \ldots, T(\mathcal{F}_{|\mathcal{F}\_|}), T(x_1), \ldots, T(x_m)\rangle$
  - $T(\mathcal{F}_i) = (S, \Sigma, \Delta, \hat{s})$, *where:*
    - $S = \{s_0, \ldots, s_{|\mathcal{F}_i|}\}$
    - $\Sigma = \{ev(\mathcal{F}_{i,j}) \mid j \in \{1 \ldots |\mathcal{F}_i|\}\} \cup \{\tau\}$
    - $\Delta = \{(s_0, \tau, s_0)\} \cup \{(s_0, ev(\mathcal{F}_{i,j}), s_j) \mid j \in \{1 \ldots |\mathcal{F}_i|\}\}$
    - $\hat{s} = s_0$
  - $T(x_i) = (S, \Sigma, \Delta, \hat{s})$, *where:*
    - $S = \{s_0, s_1, s_2\}$
    - $\Sigma = \{\tau, true_i, false_i\}$
    - $\Delta = \{(s_0, \tau, s_1), (s_0, \tau, s_2), (s_1, true_i, s_1), (s_2, false_i, s_2)\}$
    - $\hat{s} = s_0$
- $\mathcal{R} = \bigcup_{i \in \{1 \ldots |\mathcal{F}\_|\}} \{ev(\mathcal{F}_{i,j})^{\{l, |\mathcal{F}\_|+i\}} | j \in \{1 \ldots |F_i|\}\}$
  $\cup \{\tau^{\{i\}} | i \in \{1 \ldots |\mathcal{F}\_|\}\}$
- *The function $ev$ maps a literal to an event as follows.*
  - $ev(\neg x_i) = false_i$
  - $ev(x_i) = true_i$
- $e^I$ *gives rise to the rule $(t, e)$ where $t_i = e$ if $i \in I$, and $t_i = -$ if $i \notin I$.*

Even though we do not explicitly present an algorithm for the $Translate$ function, from its definition, it is easy to devise a simple procedure that constructs the supercombinator by traversing the formula, and that takes $\mathcal{O}(|\mathcal{F}| + m)$ time for this. We discuss this procedure as follows.

- $\mathcal{S}$ can be constructed in $\mathcal{O}(|\mathcal{F}| + m)$ steps. This follows from the fact that each $T(\mathcal{F}_i)$ and $T(x_i)$ can be constructed in $\mathcal{O}(|\mathcal{F}_i|)$ and $\mathcal{O}(1)$, respectively.

    - One can construct the LTS $T(\mathcal{F}_i)$ in $\mathcal{O}(|\mathcal{F}_i|)$ steps. It is easy to see that each element in such an LTS can be constructed by a single traversal of the clause, that is, in $\mathcal{O}(|\mathcal{F}_i|)$ steps.
    - One can construct $T(x_i)$ in $\mathcal{O}(1)$. This follows from the fact that each element of this LTS can be constructed in constant time.
- $\mathcal{R}$ can be constructed in $\mathcal{O}(|\mathcal{F}|)$. This follows from the fact that each literal gives rise to a rule, and a rule is formed in constant time. Also, $|\mathcal{F}\_|$ $\tau$-rules are created, but note that $|\mathcal{F}| \geq |\mathcal{F}\_|$.

After the presentation of our mapping, we show that our pair-candidate detection problem can be soundly used to check the satisfiability of a CNF boolean function. To this end, we show that there exists a satisfying assignment for a given formula if and only if there exists a Pair deadlock candidate for the translated supercombinator.

**Theorem 3.** *Let $\mathcal{F}(x_1,\ldots,x_m)$ be a CNF boolean function with $m$ variables, and $\mathcal{S}$ a supercombinator such that $\mathcal{S} = Translate(\mathcal{F},(x_1,\ldots,x_m))$. Also, let $(S,\Sigma,\Delta,\hat{s})$ be the induced supercombinator of $\mathcal{S}$.*

$$\exists\, x_1,\ldots,x_m : \mathbb{B} \bullet \mathcal{F}(x_1,\ldots,x_m) \Longleftrightarrow \exists\, s : S \bullet pair\_candidate(s)$$

*Proof. Case 1 ($\Longrightarrow$).* To prove this case, given the existence of a model for the formula $\mathcal{F}$, we show that there exists a state $s$ which represents a Pair deadlock candidate. Let us assume that $\mathcal{M} = (\mathcal{M}_1,\ldots,\mathcal{M}_m)$ is the model in question, where $\mathcal{M}_i$ is either $x_i$ or $\neg x_i$, according to whether $x_i$ has been assigned to $true$ or $false$, respectively.

1. In order to be a model for $\mathcal{F}$, there must exist a literal satisfied in each clause. So, we assume, without loss of generality, that the $k(i)$-th literal of the $i$-clause is satisfied, i.e. $\mathcal{F}_{i,k(i)} = \mathcal{M}_{var(i,k(i))}$.
2. Based on this, we can form the following supercombinator state $s$ where
   (a) $s_i = s_{k(i)}$ if $i \in \{1 \ldots |\mathcal{F}\_|\}$
   (b) $s_i = s_1$ if $i \in \{|\mathcal{F}\_|+1 \ldots |\mathcal{F}\_|+m\}$ and $\mathcal{M}_j = x_j$, with $j = i - |\mathcal{F}\_|$
   (c) $s_i = s_2$ if $i \in \{|\mathcal{F}\_|+1 \ldots |\mathcal{F}\_|+m\}$ and $\mathcal{M}_j = \neg x_j$, with $j = i - |\mathcal{F}\_|$
   This state symbolises that the clauses components have reached satisfying states by reaching their respective terminal state $s_{k(i)}$, in which the literal $\mathcal{F}_{i,k(i)}$ has been satisfied, and that the variable components are in states miming the model $\mathcal{M}$.
3. $s$ is blocked thanks to two facts. First, all states of clause components in $s$ are terminal, i.e. they cannot engage in any more actions. Second, from our translation, all the rules involve the participation of one of these components. So, based on this two facts, no rule can be applied.
4. $s$ is pairwise reachable. First, a pair of states involving two clause component states or two variable component states are trivially pairwise reachable, as they are in interleaving. Moreover, a pair of component states involving the component state $s_j$ of clause $i$ and the component state $s_k$ of variable $x_l$ is trivially pairwise reachable if $var(i,j) \neq l$, as this clause component can

reach $s_j$ without synchronising with this variable component. So, the only case left to prove is the case involving the component state $s_j$ of clause $i$ and the component state $s_k$ of variable $x_l$ where $var(i, j) = l$.
In such a case, as we know that $s_j$ is a terminal state and assuming that $\mathcal{F}_{i,j} = x_l$ (the other case where $\mathcal{F}_{i,j} = \neg x_l$ is symmetric), we can deduce that $(s_0, true_l, s_j) \in \Delta_i$. Also, thanks to $\mathcal{F}_{i,k(i)} = \mathcal{M}_{var(i,k(i))}$, we know that $s_k = s_1$. From our translation, we also know that $(s_0, \tau, s_1), (s_1, true_l, s_1) \in \Delta_k$. Thus, the sequence of event $\langle \tau, true_l \rangle$ leads these components pairwise projection from their initial state to the desired state $(s_j, s_k)$, thereby making this pair of component states pairwise reachable.

*Case 2 ($\Longleftarrow$).* In this case, we show that given a pairwise-deadlocked state $s$ for the translated supercombinator, we can construct a model $\mathcal{M}$.

1. A blocked state must have all clause components in a terminal state. Otherwise, a clause component would be in the initial state, thereby being able to perform the transition $(s_0, \tau, s_0)$ on its own. So, we assume, without loss of generality, that for the clause component $i$, its terminal state reached is $s_{k(i)}$.
2. From this fact, we have that each clause component $i$ must have performed a transition $(s_0, ev(\mathcal{F}_{i,k(i)}), s_{k(i)})$.
3. From the pairwise-reachability requirement, we know that each variable component present in one of these satisfied literal, namely $x_{var(i,k(i))}$, must be in a state such that it agrees to perform $ev(\mathcal{F}_{i,k_i})$. Hence, they must be either in state $s_1$ or $s_2$, according to whether $\mathcal{F}_{i,k_i} = x_{var(i,k(i))}$ or $\mathcal{F}_{i,k_i} = \neg x_{var(i,k(i))}$, respectively.
4. Based on these facts, we can construct the model $\mathcal{M}$ where for each clause index $i$, $\mathcal{M}_{var(i,k(i))} = \mathcal{F}_{i,k(i)}$.
5. $\mathcal{M}$ represents a model to $\mathcal{F}$, as each clause $\mathcal{F}_i$ has the literal $\mathcal{F}_{i,k(i)}$ satisfied. $\square$

Most of the incomplete techniques for deadlock analysis rely on polynomial-time checkable conditions; ours, however, is based on a NP-complete problem. This suggests that our strategy deals with a different class of systems for which deadlock analysis seems to be a more difficult task.

## 6 Accuracy of the Pair Framework

In this section, we shed light on the class of systems that can be successfully proved deadlock free by Pair. To this end, we analyse its generality by comparing it to the SDD framework. In this comparison, we first outline the class of systems tackled by SDD and then we show that our approach can tackle a class of systems strictly larger than SDD.

The SDD framework has been able to successfully prove deadlock freedom for some relevant classes of system. Martin has shown that his framework can prove deadlock freedom for systems designed using two well-known design rules: the *resource-allocation* and the *client-server*. The resource allocation rule has been proposed initially as a mechanism for avoinding deadlocks when allocating the resources of an operating system to programs [8], whereas client-server protocols constitutes a very common paradigm for the interaction of distributed system. Both rules prevent an undesired cycle of ungranted requests from arising.

### 6.1 Pair is at least as good as SDD

A deadlocked state has to exhibit a cycle of ungranted requests between component states when components are deadlock-free and termination-free. So, to compare Pair with SDD, we limit ourselves to such systems in this section.

In this restricted setting, we show that our approach can prove deadlock freedom for a system whenever SDD can. This follows from the claim that for a live system, a blocked state must exhibit a cycle of ungranted requests.

**Lemma 2 (Theorem 1 in [14]).** *Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $\mathcal{U}$ the disjoint union of all the component states of each component.*

$$\exists\, s : S \bullet \mathit{blocked}(s) \implies \exists\, c : \mathcal{U}^* \bullet \mathit{sdd\_candidate}(c)$$

**Theorem 4.** *Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $\mathcal{U}$ the disjoint union of all the component states of each component.*

$$\neg\, \exists\, c : \mathcal{U}^* \bullet \mathit{sdd\_candidate}(c) \implies \neg\, \exists\, s : S \bullet \mathit{pair\_candidate}(s)$$

*Proof.* From Lemma 2, we can deduce the following: $\exists\, s : S \bullet \mathit{pair\_candidate}(s) \implies \exists\, c : \mathcal{U}^* \bullet \mathit{sdd\_candidate}(c)$. Our claim follows easily from this result.

### 6.2 Pair is more accurate than SDD

Even though SDD is accurate for a reasonably large and relevant class of systems, it is unable to prove deadlock freedom for non-hereditary deadlock-free systems. This is shown by Lemma 2: if a subsystem deadlocks then there must exist a cycle of ungranted requests between the states of components in this subsystem that constitutes a SDD candidate. Roughly speaking, SDD can be seen as a method that tries to prove *hereditary* deadlock freedom using local analysis. On the other hand, our method can prove deadlock freedom for both hereditary and non-hereditary deadlock-free systems, such as the following example.

*Example 2.* This well-known example system is composed of three different components: forks, philosophers and a butler. We parametrise our system with $N$, which denotes the number of philosophers in the system.

A philosopher has access to a table at which it can pick up two forks to eat: one at its left-hand side and the other at its right-hand side. A fork is placed, and shared, between philosophers sitting adjacently in the table. The behaviour of philosopher (fork) $i$ is depicted in Figure 2 (4). $\oplus$ stands for addition modulo $N$.

Given that these components synchronise in their shared events, the philosophers and forks can reach a deadlock state in which all philosophers have acquired

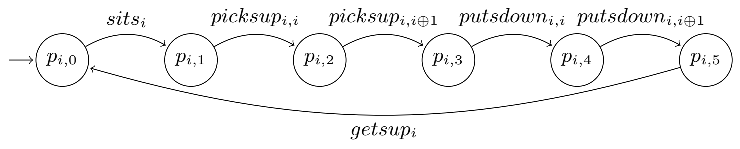

**Fig. 3.** LTS of philosopher $i$.

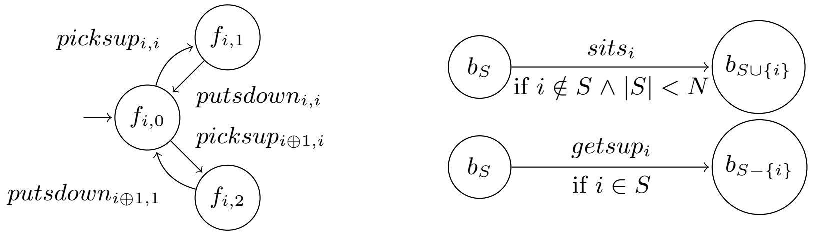

**Fig. 4.** LTS of fork $i$ and transitions of the butler process.

their left-hand side forks and, as a consequence, no right-hand side fork is left to be acquired. The butler is introduced to prevent all the philosophers from sitting at the table at the same time, thereby precluding this deadlock state. We use $b_S$ to depict the state in which the butler has allowed the philosophers in $S$ to the table. So, the butler states space is given by the set of all $b_S$ where $S \in \mathbb{P}(\{1 \dots N\}) - \{1 \dots N\}$. Its transitions are created as depicted in Figure 4, and its initial state is given by $b_\emptyset$.

The complete system has $N$ philosophers, $N$ forks and a butler, and these components synchronise on their shared events. Despite being deadlock free, this system has a cycle of component states that forms a SDD candidate, namely, where all the philosphers have acquired their left-hand fork:

$$\langle p_{0,2}, f_{1,1}, p_{1,2}, f_{2,1}, \dots, p_{N-2,2}, f_{N-1,1}, p_{N-1,2}, f_{0,1}\rangle$$

However, this SDD candidate cannot be extended to a pair candidate, because the latter would have to include a butler state, and no butler state is consistent with this combination of philosopher states. □

This example shows that the Pair method is strictly more accurate than SDD. Going a step further, one can perceive this example as depicting a class of non-hereditary deadlock-free systems in which a guard-like component prevents a subsystem's deadlock state from being reached. In our example, the deadlocking subsystem consists of the composition of philosophers and forks, and the butler is the guard-like component leading the system away from the deadlocked state. As for the relevance of this class of systems, many parallel systems make use of semaphores, which are guard-like components, to avoid undesired concurrent behaviour (such as reaching a deadlocked state).

Moreover, our method has better accuracy than SDD even for hereditary deadlock-free systems, thanks to the fact that we use local reachability and blocking information to its full extent. This increase in accuracy, however, comes with a price. The explicit exploration of, only, localised state spaces helps to tame the complexity of checking our deadlock-freedom condition. Nevertheless, by strengthening the candidate's definition in relation to prior techniques, we end up with a NP-complete problem.

## 7 Pair Candidate Detection using a SAT Solver

In this section, we propose a procedure that encodes the pair-candidate detection problem in terms of propositional satisfiability, which can later be checked by a SAT solver. Given a supercombinator machine as an input, our procedure creates a propositional formula in conjunctive normal form (CNF). A satisfying assignment for this formula gives a pair candidate: the variables assigned to true correspond to a combination of component states that forms a pair candidate, whereas a proof of unsatisfiability entails deadlock freedom for the input system. The use of intermediate structures in our encoding procedure and the application of a SAT solver in the process of deadlock checking was inspired by the success of the SLAP tool [16], which uses SAT solvers for the verification of livelocks[4].

We consider for the sake of presentation that we are translating the supercombinator machine $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. Additionally, we assume component states are unique across the system and that $s_{i,j}$ denotes the state $j$ of the component $i$. Our encoding procedure can be divided into two parts: an initial one where intermediate structures are calculated from the supercombinator machine, and a final one where the boolean formula is generated based on these intermediate structures.

The intermediate structures can be seen as storing information that is later used to filter out combinations of component states that do not belong to a valid pair candidate. The first intermediate structure created, $\mathit{RequireSync}_i$, stores for each component the states in which cooperation is required. So, it provides information to filter out component states that can independently act and are, therefore, trivially not blocked.

**Definition 11.** $\mathit{RequireSync}_i = \{s | s \in S_i \land \neg \mathit{independent}_i(s)\}$

$$- \ \mathit{independent}_i(s) \mathrel{\widehat=} \exists (e, a) : \mathcal{R} \bullet (e_i \neq - \land \forall k : \{1 \ldots n\} \mid k \neq i \bullet e_k = -) \\ \land (\exists s' : S_i \bullet (s, e_i, s') \in \Delta_i)$$

The structure $\mathit{CanSync}$ stores blocking information about pairs of components. It provides information to filter out pairs of component states in which components can interact. The triple disjointness assumption means that this pairwise information is enough to determine whether a system state is blocked.

**Definition 12.**

[4] There are some significant differences with SLAP: here the propositional formula is satisfied by a possible deadlock, whereas in SLAP the propositional formula is satisfied by a *proof of livelock freedom*. We might also note that livelock arises from a sequence of states, whereas deadlock arises in a single one.

$$CanSync = \bigcup_{i,j \in \{1 \ldots n\} \wedge i \neq j} \left\{ (s, s') \,\middle|\, \begin{array}{l} s \in RequireSync_i \wedge s' \in RequireSync_j \wedge \\ reachable_{i,j}((s, s')) \wedge sync_{i,j}(s, s') \end{array} \right\}$$

– $sync_{i,j}(s, s') = \exists (e, a) : \mathcal{R} \, ; t : S_i \, ; t' : S_j \bullet (s, e_i, t) \in \Delta_i \wedge (s', e_j, t') \in \Delta_j$

The last structure $NPR$ (Not Pairwise Reachable) collects local reachability properties and is used to filter out pairs of components that are not mutually reachable.

**Definition 13.**

$$NPR = \bigcup_{i,j \in \{1 \ldots n\} \wedge i \neq j} \left\{ (s, s') \,\middle|\, \begin{array}{l} s \in RequireSync_i \wedge s' \in RequireSync_j \wedge \\ \neg reachable_{i,j}((s, s')) \end{array} \right\}$$

In the second phase of our encoding procedure, we construct a boolean formula based on these derived structures. The formula generated is a conjunction of three constraints; each of them uses the information encompassed in a derived structure to filter out invalid combinations of component states. For the construction of our formula, we use our state representation $s_{i,j}$ to denote the boolean variable representing this state. So, the assignment $s_{i,j} = true$ might be seen as claiming that this state belongs to a pair candidate, whereas $s_{i,j} = false$ means it does not.

The first constraint, *State*, restricts the space of valid combinations of component states to complete snapshots. As discussed, only states in *RequireSync* structure are relevant.

**Definition 14.**

$$State \mathrel{\widehat{=}} \bigwedge_{i \in \{1 \ldots n\}} \Big( \bigvee_{s \in RequireSync_i} s \Big) \wedge \bigwedge_{i \in \{1 \ldots n\}} \Big( \bigwedge_{s, s' \in RequireSync_i \wedge s \neq s'} (\neg s \vee \neg s') \Big)$$

The second constraint restricts the space of valid combinations of component states to the ones respecting local reachability properties.

**Definition 15.** $Reachable \mathrel{\widehat{=}} \bigwedge_{(s,s') \in NPR} (\neg s \vee \neg s')$

Finally, the last constraint ensures that the space of valid combinations of component states are the ones respecting our blocking requirement.

**Definition 16.** $Blocked \mathrel{\widehat{=}} \bigwedge_{(s,s') \in CanSync} (\neg s \vee \neg s')$

The validity of this encoding is based on the following theorem.

**Theorem 5.** *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $\mathcal{F}$ be the boolean function with $m$ variables whose formula is the result of applying our encoding procedure to $\mathcal{S}$. The following holds.*

$$\exists\, s : S \bullet pair\_candidate(s) \Leftrightarrow \exists\, x_0, \ldots, x_m : \mathbb{B} \bullet \mathcal{F}(x_0, \ldots, x_m)$$

*Proof.* This can be deduced from the correspondence between the constraints in our formula and the aspects that they encode.

*Case 1 ($\Longrightarrow$).* In this case, we assume $s$ to be a pair candidate, and we show that the assignment $m$ where all the variables (i.e. component states) but the ones in $s$ are set to false is a model for $\mathcal{F}$.

First of all, we know that $s$ has no granted states, otherwise $s$ would not be blocked, thus, not a pair candidate.

- $m$ satisfies the *State* constraint. The state constraint mandates the model to have a single component state of each component assigned to true and that is clearly the case for $m$, as it is derived from the system state $s$.
- $m$ satisfies the *Reachability* constraint. As $s$ is pairwise reachable, all the pairs of component state in $s$ must be reachable in their pairwise projection of $\mathcal{S}$. Hence, no two component states assigned to *true* in $m$ form a pair in $NPR$. As a consequence, this constraint is satisfied by $m$.
- $m$ satisfies the *Blocked* constraint. As $s$ is blocked, all the pairs of component state in $s$ must not be able to communicate. Note that, we do not need to cover triples or larger combinations of interactions because our supercombinator only allows the interaction of pairs of components at a time. Hence, as assignment $m$ has no pair of component states which can interact nor granted states, the *Blocked* constraint is satisfied.

*Case 2.* In this case, we show that a model $m$ for the formula gives rise to a state of the system which represents a pair candidate. Let $s$ be a sequence of component states, ordered according to the component's orders in $\mathcal{S}$, that are assigned to *true* in the model $m$.

- $s$ is a system state. From satisfying the *State* constraint, we know that $m$ has exactly one component state of each component assigned to *true*.
- $s$ is pairwise reachable. From satisfying the *Reachable* constraint, we know that all pairs of component states in $m$ assigned to *true*, must be reachable in their corresponding supercombinator pairwise projection.
- $s$ is blocked. From the *Blocked* constraint and as we know that $m$ has only pairwise reachable component states, we know that the pair of component states in $m$ must not be able to communicate. As the granted states are not part of this formula and as interactions involving more than two components are not allowed by our supercombinator definition, we know that $s$ cannot transition to another state.

□

## 8 Practical evaluation

In this section, we evaluate our framework in practice; we modified FDR3 to generate our SAT encoding which is then checked by the Glucose 4.0 solver [6]. Our prototype and the models used in this section are available at [1]. We conduct two experiments in this section: the first one evaluates deadlock freedom for randomly generated systems, the second one evaluates deadlock freedom for some deadlock-free benchmark problems. The experiments were conducted in a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM, and the Fedora 20 operating system. In these experiments, we compare our prototype with the Deadlock Checker [15] and FDR3's deadlock freedom assertion [9]. Deadlock Checker implements the $SDD$ framework, whereas FDR3 is a complete method that performs explicit space exploration. When appropriate, we combine FDR3's explicit state exploration with partial order reduction (FDRp) [10] or compression techniques (FDRc) [18].

In the first experiment, we verify models of live systems randomly generated, but with fixed communication topologies. We verify systems whose communication topologies are grid-like, fully connected, or a pair of rings. The parameter $N$ is related to the size of these systems. The choice of these communication topologies was based on the fact that most of CSP benchmark problems use one of these or a variation. For each of topology and $N$, we generate 900 random systems.

In Table 1, we summarise the accuracy results obtained. For the accuracy comparison, we take FDR3's deadlock assertion out, as it is a complete method. Also, the reason why we present in some occasions the absolute number of deadlock-free systems is that we use FDR3 to get the exact number of deadlock-free systems, but when FDR3 times out, this number is unavailable. In Table 2, for FDR3, we present the figures for the method that worked best. So, for the pair of rings, applying partial order reduction made FDR3 scale better, whereas for the other two cases, explicit state exploration was the best option.

Based on the data gathered in this first experiment, we can conclude that our prototype provides a far better compromise between accuracy and speed than the Deadlock Checker for the systems checked. The fact that hereditary deadlock freedom is more difficult to achieve than deadlock freedom seems to be the reason why our approach is substantially more accurate. In terms of efficiency, we see that our method scales fairly well for the generated systems. It fared better than

|  | Rings |  | Grid |  | Fully |  |
|---|---|---|---|---|---|---|
| N | Pair | SDD | Pair | SDD | Pair | SDD |
| 3 | 99.13 | 64.34 | 100 | 34.44 | 93.98 | 18.67 |
| 4 | 99.67 | 68.19 | (599) | (106) | 98.76 | 6.4 |
| 5 | 99.71 | 73.57 | (635) | (96) | 98.11 | 1.8 |
| 6 | 98.98 | 77.41 | (644) | (92) | 99.25 | 1.1 |
| 7 | 100 | 76.14 | (771) | (30) | 99.28 | 0.1 |
| 8 | (469) | (385) | (773) | (57) | 99.65 | 0 |
| 9 | (500) | (422) | (779) | (28) | 99.83 | 0 |
| 10 | (517) | (444) | (774) | (8) | 99.52 | 0 |
| 15 | (590) | (491) | (900) | (0) | (692) | (0) |
| 20 | (645) | (547) | (900) | (0) | (703) | (0) |
| 25 | (680) | (566) | (887) | (0) | (742) | (0) |

**Table 1.** Accuracy comparison; the numbers not in parentheses depict the percentages of deadlock free systems proved as so. The numbers in parentheses represent the total number of deadlock free systems proved as so.

FDR3 even when combined with sophisticated techniques to combat the state space explosion problem. For most of the cases, our method also fared better than the Deadlock Checker. For the cases in which the Deadlock Checker scales better, we can see a considerable difference in the accuracy of the two methods that justifies the difference in their speed.

Our second experiment consists of applying deadlock verification methods to some benchmark problems that are carefully designed to be deadlock free. We chose six benchmark problems that are proved deadlock free by Pair. These problems are the alternating-bit protocol (ABP), the sliding window protocol (SWP), a binary telephone switch (Telephone), the mad postman routing algorithm (Routing), the asymmetric dining philosophers (Phils), and the butler solution to the dining philosophers (Butler). These problems are discussed in detail in [19]. For each of these benchmarks problems, we vary a parameter $N$ which relates to the size of these systems. Table 3 presents the results of this second experiment, which suggests that our method scales similarly to the combination of FDR3's assertion techniques with compression techniques. We point out that the effective use of compression techniques requires a careful and skilful application of those, whereas our method is fully automatic. In fact, our strategy seems to be the most efficient option for all but the Routing problem in which both the Deadlock Checker and FDR3's assertion with compression techniques outperform us.

Unsurprisingly, for some other benchmark problems our method is not able to prove deadlock freedom. The reason is that, for these cases, deadlock freedom depends on some global invariant preserved by the system (or perhaps by larger subsets of the system than the pairs used here), and as argued, this type of reasoning is beyond the capabilities of our method. For instance, proving deadlock freedom for the Milner's scheduler problem, which is a fairly simple benchmark problem, is out of our method's reach. The issue with Milner's scheduler is that it is essentially a token ring which depends on the fact that there is always precisely one token present; this latter property cannot be proved by local analysis.

| | Rings | | | Grid | | | Fully | | |
|---|---|---|---|---|---|---|---|---|---|
| N | Pair | SDD | FDR3p | Pair | SDD | FDR3 | Pair | SDD | FDR3 |
| 3 | 37.38 | 66.04 | 40.91 | 40.47 | 71.01 | 70.27 | 37.39 | 65.64 | 42.74 |
| 4 | 37.88 | 67.65 | 42.89 | 44.89 | 76.57 | * | 39.04 | 70.02 | 43.36 |
| 5 | 39.00 | 68.30 | 51.60 | 52.67 | 90.50 | * | 39.74 | 74.19 | 43.97 |
| 6 | 39.67 | 69.97 | 103.83 | 60.85 | 104.07 | * | 42.46 | 83.18 | 48.96 |
| 7 | 41.07 | 71.69 | 788.03 | 70.39 | 113.95 | * | 45.50 | 92.91 | 61.47 |
| 8 | 41.12 | 73.11 | * | 84.67 | 128.41 | * | 49.24 | 103.08 | 118.78 |
| 9 | 41.90 | 73.71 | * | 101.18 | 142.65 | * | 53.91 | 115.87 | 415.87 |
| 10 | 42.67 | 75.31 | * | 124.80 | 157.76 | * | 60.32 | 125.60 | 1897.71 |
| 15 | 46.75 | 80.52 | * | 326.56 | 249.27 | * | 108.99 | 210.65 | * |
| 20 | 52.09 | 89.03 | * | 797.25 | 385.99 | * | 208.37 | 372.44 | * |
| 25 | 57.48 | 95.74 | * | 1745.72 | 566.27 | * | 382.89 | 645.74 | * |

**Table 2.** Efficiency comparison; we measure in seconds the time taken to check deadlock freedom for the 900-systems batch, and * means that the methods has timed out. We establish a time out of 2000 seconds for checking each batch.

## 9 Conclusion

We have introduced a new test for deadlock freedom that extends the capabilities of current state-of-the-art incomplete approaches. We give up completeness to achieve scalability. Our intention is to propose a method rivalling the speed of current incomplete approaches but with a considerable increase in accuracy. To do so, we introduce a stronger deadlock candidate definition, which allows the analysis of both hereditary and non-hereditary deadlock-free systems, and we bring the power of SAT checking to bear on a style of local analysis of systems that reaches back decades. Our ambition is to have a deadlock checker which can be used as a matter of course on systems developed by non-experts who do not necessarily have any knowledge of established design patterns for deadlock freedom, such as those previously proposed by both the authors.

For the systems tested, our strategy seems to provide a better compromise between speed and accuracy. It appears to perform strongly in terms of speed when

| | ABP | | | |
|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3por |
| 3 | 0.05 | 0.09 | 0.04 | 0.05 |
| 4 | 0.06 | 0.09 | 0.04 | 0.05 |
| 5 | 0.05 | 0.11 | 0.04 | 0.05 |
| 10 | 0.06 | 0.12 | 0.04 | 0.06 |
| 15 | 0.07 | 0.14 | 0.04 | 0.07 |
| 30 | 0.06 | 0.21 | 0.05 | 0.44 |
| 50 | 0.16 | 0.29 | 0.05 | 0.12 |

| | SWP | | | | |
|---|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3c | FDR3por |
| 3 | 0.29 | 0.88 | 0.14 | 0.24 | 0.21 |
| 4 | 2.83 | 40.83 | 0.58 | 1.13 | 3.57 |
| 5 | 42.79 | * | 3.23 | 4.62 | * |
| 6 | * | * | 18.38 | 25.25 | * |
| 7 | * | * | * | * | * |
| 8 | * | * | * | * | * |
| 9 | * | * | * | * | * |

| | Telephone | | | | |
|---|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3c | FDR3por |
| 3 | * | - | 0.06 | 0.17 | * |
| 4 | * | - | 0.11 | 2.93 | * |
| 5 | * | - | 0.32 | * | * |
| 6 | * | - | 1.34 | * | * |
| 7 | * | - | 6.27 | * | * |
| 8 | * | - | 31.68 | * | * |
| 9 | * | - | * | * | * |

| | Routing | | | | |
|---|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3c | FDR3por |
| 3 | * | 0.10 | 0.06 | 0.10 | * |
| 4 | * | 0.11 | 0.09 | 0.14 | * |
| 5 | * | 0.13 | 0.13 | 0.18 | * |
| 10 | * | 0.30 | 0.99 | 0.71 | * |
| 20 | * | 1.11 | 13.27 | 4.45 | * |
| 30 | * | 3.30 | * | 16.72 | * |

| | Phils | | | | |
|---|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3c | FDR3por |
| 3 | 0.06 | 0.13 | 0.05 | 0.08 | 0.07 |
| 4 | 0.07 | 0.13 | 0.05 | 0.09 | 0.07 |
| 5 | 0.07 | 0.13 | 0.05 | 0.09 | 0.07 |
| 10 | 122.15 | 0.13 | 0.06 | 0.14 | 0.42 |
| 25 | * | 0.17 | 0.08 | 0.40 | * |
| 50 | * | 0.23 | 0.13 | 1.64 | * |
| 100 | * | 0.36 | 0.31 | 15.83 | * |

| | Butler | | | | |
|---|---|---|---|---|---|
| N | FDR3 | SDD | Pair | FDR3c | FDR3por |
| 3 | 0.06 | - | 0.06 | 0.09 | 0.06 |
| 4 | 0.07 | - | 0.6 | 0.10 | 0.07 |
| 5 | 0.26 | - | 0.6 | 0.10 | 0.43 |
| 6 | 0.11 | - | 0.7 | 0.12 | 0.08 |
| 7 | 0.32 | - | 0.9 | 0.14 | 0.13 |
| 8 | 1.91 | - | 0.12 | 0.17 | 0.22 |
| 9 | 16.80 | - | 0.19 | 0.22 | 0.52 |

**Table 3.** Benchmark efficiency comparison. We measure in seconds the time taken to check deadlock freedom for each system. * means that the methods has timed out; we establish a time out of 40 seconds for checking each system. - means that the method is unable to prove deadlock freedom for the system.

compared to SDD, compression and partial order techniques. As for accuracy, our method is strictly more accurate than SDD, and in particular, it is able to tackle non-hereditary deadlock-free system, a class of systems neglected by most incomplete techniques. Nevertheless, our technique (and any other one that uses local analysis) cannot prove deadlock freedom for systems in which this property is guaranteed by some invariant on the global behaviour of systems.

As a future work, we plan to improve accuracy, without excessively damaging speed, by proposing methods to efficiently calculate some global invariants. This should not make our method complete, but it should enable the handling of systems which are deadlock free by some global property of the system.

## Acknowledgments

We are grateful to Jöel Ouaknine and James Worrell for many fruitful discussions concerning this work. The first author is a CAPES Foundation scholarship holder (Process no: 13201/13-1). The second and third authors are partially sponsored by DARPA under agreement number FA8750-12-2-0247.

## References

1. Pedro Antonino, A. W. Roscoe, and Thomas Gibson-Robinson. Experiment package, 2015. `http://www.cs.ox.ac.uk/people/pedro.antonino/exp.zip`.
2. Pedro Antonino, Augusto Sampaio, and Jim Woodcock. A refinement based strategy for local deadlock analysis of networks of CSP processes. In *FM*, volume 8442 of *LNCS*, pages 62–77, 2014.
3. Pedro R.G. Antonino, Marcel Medeiros Oliveira, Augusto C.A. Sampaio, Klaus E. Kristensen, and Jeremy W. Bryans. Leadership election: An industrial SoS application of compositional deadlock verification. In *NFM*, volume 8430 of *LNCS*, pages 31–45, 2014.
4. Paul C. Attie, Saddek Bensalem, Marius Bozga, Mohamad Jaber, Joseph Sifakis, and Fadi A. Zaraket. An Abstract Framework for Deadlock Prevention in BIP. In *Formal Techniques for Distributed Systems*, number 7892 in Lecture Notes in Computer Science, pages 161–177. Springer, 2013.
5. Paul C. Attie and Hana Chockler. Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In *VMCAI*, pages 465–481. Springer, 2005.
6. Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009.
7. Stephen D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, **4**:209–230, 1991.
8. Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, **3**(**2**):67–78, 1971.
9. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In *TACAS*, volume 8413 of *LNCS*, pages 187–201, 2014.
10. Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 188–203. Springer International Publishing, 2015.

11. Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, **2(2)**:149–164, 1993.
12. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
13. Christian Lambertz and Mila Majster-Cederbaum. Analyzing Component-Based Systems on the Basis of Architectural Constraints. In *Fundamentals of Software Engineering*, pages 64–79. Springer, April 2011.
14. Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems.* PhD thesis, University of Buckingham, 1996.
15. J.M.R. Martin and S.A. Jassim. An efficient technique for deadlock analysis of large scale process networks. In *FME '97*, pages 418–441, 1997.
16. Joël Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. A static analysis framework for livelock freedom in CSP. *Logical Methods in Computer Science*, **9(3)**, 2013.
17. A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Inf. Comput.*, **75(3)**:289–327, 1987.
18. A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check $10^{20}$ dining philosophers for deadlock. In *TACAS*, pages 133–152, 1995.
19. A.W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.