Efficient Deadlock-Freedom Checking using Local Analysis and SAT Solving

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, the so-called non-hereditary deadlock-free systems (i.e. deadlock-free systems that have a deadlocking subsystem), 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 is usually an important goal when developing and verifying a concurrent system. A system is deadlock free if and only if it cannot reach a state in which it can perform no further actions. Moreover, many safety properties can be reduced to verifying deadlock freedom of modified systems [12]. Unsurprisingly, even when restricted to deadlock analysis, existing automated verification techniques still suffer from the state explosion problem.

Incomplete techniques for deadlock analysis [6, 15, 14] 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 typically 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

[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.

a component to another if and only if the former is trying to communicate with the 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 construct (either explicitly or implicitly) 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 present these cycles and, as such, they cannot be handled by these methods. For instance, *non-hereditary* deadlock-free systems, namely, deadlock-free systems that have a subsystem that can deadlock, cannot be tackled by current techniques using local analysis. Secondly, to keep the analysis of these dependency graphs efficient, some local properties, which could be used to improve 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 these sorts of imprecision. 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 [15] (see Section 3), and by the successful use of SAT checkers for livelock analysis reported in [17].
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 accuracy of our method. In the following section, we give an encoding of our deadlock-freedom analysis as a SAT problem. Section 7 presents some experiments conducted to assess the accuracy and efficiency of our framework. Finally, in Section 8, we present our concluding remarks.

2 Background

Communicating Sequential Processes (CSP) [13, 20] is a notation used to model concurrent systems where processes interact, exchanging messages. Here we describe some structures used by the refinement checker FDR3 [10] 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 [20].

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;*
- Σ *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 τ (the silent event) and $\checkmark$ (the termination signal) are considered members of Σ, since there is no difference between them and regular events in the context 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 (Structural Operational Semantics) style, FDR3 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 FDR3'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.*
    - $triple\_disjoint(e) \mathrel{\widehat{=}} \forall i, j, k : \{1 \ldots n\} \mid i \neq j \land j \neq k \land i \neq k \bullet$
      $$e_i = - \lor e_j = - \lor e_k = -$$
  - $a \in \Sigma$ *is the event the supercombinator performs.*

This restriction is similar to ones adopted in related work to ours [15, 6]. Henceforth, we omit the mention of triple-disjoint.

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

[2] FDR3 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:*

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

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:*

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

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 with finite alphabets.

3 Related Work

Two of the authors of this paper have previously investigated the role played by local analysis in establishing deadlock freedom in [18, 8, 1, 4]. 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 [6, 5, 14, 15], fully-automated but incomplete techniques for deadlock freedom are introduced. These techniques are proposed for different contexts and types of concurrency: [6] proposes a method for analysing syntactically-restricted shared-variable concurrent programs, [5] adapts [6] to a more general setting meant to describe component-based message-passing systems, [14] proposes a method for architecturally-restricted component-based systems interacting via message passing, and [15] 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 [15]. We regard our framework as a development on the SDD. Martin's analysis of SDDs is one of the most general prior approaches to local deadlock analysis.

In that work [15], 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 where: $reachable_{i,j}$ 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* $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. *Let* $\mathcal{U}$ *be the disjoint union of all* S_i *and* $s_{i,j}$ *denotes 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 length of* c.

This method can carry out deadlock-freedom verification 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 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 framework, which we 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 can neither perform an individual event nor 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 may not prove the unreachability of a candidate.

Our framework is sound, as absence of pair candidates implies deadlock freedom. The following theorem follows from the fact that reachability implies pair-reachability. Its proof can be found in [3].

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* $s \in S$*,*

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

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

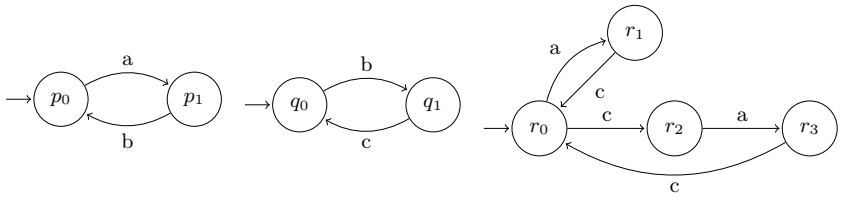

Fig. 1. LTSs of components L_1, L_2 and L_3, respectively.

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 (p_0, q_0, r_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 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 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 tackles a strictly larger class of systems.

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 *client-server* rules. The resource allocation rule has been proposed initially as a mechanism for avoiding deadlocks when allocating the resources of an operating system to programs [9], whereas client-server protocols constitute a very common paradigm for the interaction of distributed system. Both rules prevent cycles of ungranted requests from arising.

5.1 Pair is at least as good as SDD

A deadlocked state is only guaranteed to exhibit a cycle of ungranted requests if a system (or supercombinator machine) is *live*, namely all its components are deadlock-free and termination-free. So, in this section, to compare Pair with SDD, we limit ourselves to live systems.

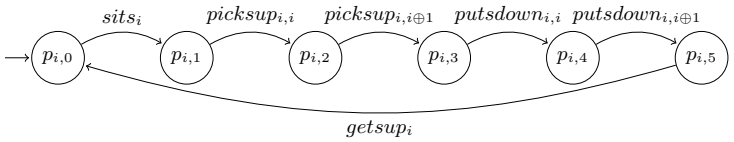

Fig. 2. LTS of philosopher i.

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 1 (Theorem 1 in [15]). *Let $\mathcal{S}$ be a live supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $\mathcal{U}$ the disjoint union of all the component states of each component.*

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

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

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

5.2 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 1: 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 (i.e. that no subsystem can deadlock) 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 (3). $\oplus$ stands for addition modulo N.

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

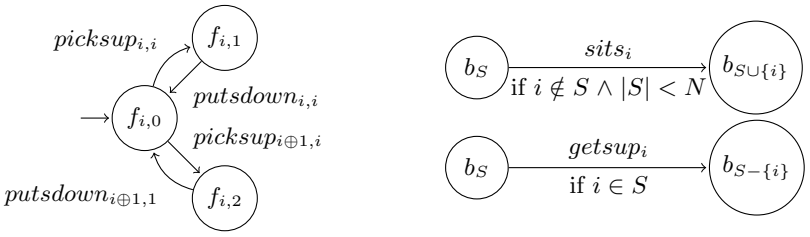

Fig. 3. LTS of fork i and transitions of the butler process.

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 3, 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, this can be seen as representative of the class of non-hereditary deadlock-free systems where one or more components prevent some subsystem's deadlock from being reached. Note that many concurrent systems use components implementing mutual exclusion algorithms or semaphores to prevent other components reaching undesired states such as a deadlock.

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 an NP-complete problem [3].

6 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 [17], 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, $RequireSync_i$, stores for each component the states in which cooperation is required. So, it provides information to filter out component states that can act independently and are, therefore, trivially not blocked.

Definition 8. $RequireSync_i = \{s | s \in S_i \wedge \neg independent_i(s)\}$

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

The structure $CanSync$ 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 9.

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

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

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

Definition 10.

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

[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.

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 11.

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

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

Definition 12. $Reachable \mathrel{\widehat{=}} \bigwedge_{(s,s') \in NPR} (\neg s \vee \neg s')$

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

Definition 13. $Blocked \mathrel{\widehat{=}} \bigwedge_{(s,s') \in CanSync} (\neg s \vee \neg s')$

7 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 [7]. Our prototype and the models used in this section are available at [2]. We describe 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 on 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 [16] and FDR3's deadlock freedom assertion [10]. 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) [11] or compression techniques (FDRc) [19].

In the first experiment, we verify models of randomly-generated live systems, but with fixed communication topologies. Our goal with this experiment is to test our tool against scripts made by non-experts. 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 many 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 sometimes present 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

	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.

	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.

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 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 four benchmark problems that are proved deadlock free by Pair. These problems are the sliding window protocol (SWP), a binary telephone switch (Telephone), the mad postman routing algorithm (Routing), and the butler solution to the dining philosophers (Butler). These problems are discussed in detail in [20]. For each of these benchmarks, 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.

	SWP				
N	FDR3	SDD	Pair	FDR3c	FDR3p
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	*	*	*	*	*

	Telephone				
N	FDR3	SDD	Pair	FDR3c	FDR3p
3	*	-	0.06	0.17	*
4	*	-	0.11	2.93	*
5	*	-	0.32	*	*
6	*	-	1.34	*	*
7	*	-	6.27	*	*
8	*	-	31.68	*	*

	Butler				
N	FDR3	SDD	Pair	FDR3c	FDR3p
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

	Routing				
N	FDR3	SDD	Pair	FDR3c	FDR3p
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	*

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.

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 of the sort we employ.

8 Conclusion

We have introduced a new test for deadlock freedom that extends the capabilities of current state-of-the-art incomplete approaches. To do so, we introduced a stronger deadlock candidate definition and we brought the power of SAT checking to bear on a style of local analysis of systems that reaches back decades. Like other incomplete methods, we sacrifice completeness to achieve scalability. This incomplete nature makes, for instance, our technique (and any other one that uses local analysis) unable to prove deadlock freedom for systems in which this property is guaranteed by some invariant on the global behaviour of systems.

Our method rivals the speed of current incomplete approaches but gives a considerable increase in accuracy. For the systems tested, it appears to perform strongly in terms of speed when 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 systems, a class of systems neglected by most incomplete techniques. Our ambition is to have a deadlock checker which can be used 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 two of the authors.

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. Additionally, we intend to extend our framework to produce counter-examples and/or other useful debugging information.

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, 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.
2. Pedro Antonino, A. W. Roscoe, and Thomas Gibson-Robinson. Experiment package, 2015. `http://www.cs.ox.ac.uk/people/pedro.antonino/exp.zip`.
3. Pedro Antonino, A.W. Roscoe, and Thomas Gibson-Robinson. Efficient deadlock analysis using local analysis and SAT solving. Tech report, University of Oxford, 2015. `http://www.cs.ox.ac.uk/people/pedro.antonino/techreport.pdf`.
4. 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.
5. Paul C. Attie, Saddek Bensalem, Marius Bozga, Mohamad Jaber, Joseph Sifakis, and Fadi A. Zaraket. An abstract framework for deadlock prevention in BIP. In *FORTE*, number 7892 in LNCS, pages 161–177. Springer, 2013.
6. Paul C. Attie and Hana Chockler. Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In *VMCAI*, pages 465–481. Springer, 2005.
7. Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009.
8. Stephen D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, **4**:209–230, 1991.
9. Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, **3**(**2**):67–78, 1971.
10. 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.
11. Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NFM*, volume 9058 of *LNCS*, pages 188–203. Springer International Publishing, 2015.
12. Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD*, **2**(**2**):149–164, 1993.
13. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.
14. Christian Lambertz and Mila Majster-Cederbaum. Analyzing Component-Based Systems on the Basis of Architectural Constraints. In *FSEN*, pages 64–79. Springer, April 2011.
15. Jeremy M. R. Martin. *The design and construction of deadlock-free concurrent systems*. PhD thesis, University of Buckingham, 1996.
16. Jeremy 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.
17. Joël Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. A static analysis framework for livelock freedom in CSP. *LMCS*, **9**(**3**), 2013.
18. A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Inf. Comput.*, **75**(**3**):289–327, 1987.
19. 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.
20. A.W. Roscoe. *Understanding concurrent systems*. Springer, 2010.