https://doi.org/10.1007/s00165-019-00483-2

Formal Aspects of Computing (2019) 31: 375–409

Formal Aspects of Computing

Efficient verification of concurrent systems using local-analysis-based approximations and SAT solving

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

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract. This work develops a type of local analysis that can prove concurrent systems deadlock free. As opposed to examining the overall behaviour of a system, local analysis consists of examining the behaviour of small parts of the system to yield a given property. We analyse pairs of interacting components to approximate system reachability and propose a new sound but incomplete/approximate framework that checks deadlock and local-deadlock freedom. By replacing exact reachability by this approximation, it looks for deadlock (or local-deadlock) candidates, namely, blocked (locally-blocked) system states that lie within our approximation. This characterisation improves on the precision of current approximate techniques. In particular, it can tackle non-hereditary deadlock-free systems, namely, deadlock-free systems that have a deadlocking subsystem. These are neglected by most approximate techniques. Furthermore, we demonstrate how SAT checkers can be used to efficiently implement our framework, which, typically, scales better than current techniques for deadlock-freedom analysis. This is demonstrated by a series of practical experiments.

Keywords: Approximate reachability, Local analysis, SAT solving, Formal verification, Deadlock freedom, Model checking, Approximate verification

1. Introduction

Fully automatic verification techniques, such as model checking, have been severely hindered by the state space explosion problem [BK08]. In the context of concurrent and distributed systems, for instance, verification techniques have to analyse a state space that typically grows exponentially on number of components in the system. This explosion makes the analysis of even moderated-sized systems infeasible. Since model checkers were invented, finding techniques to cope with this problem has been an active area of research.

Correspondence and offprint requests to: P. Antonino, E-mail: prgantonino@gmail.com; pedro.antonino@cs.ox.ac.uk

Many techniques to tackle the state-space explosion problem have been invented: most notably partial-order reductions [GW93, GRHRW15, Val92, Pel93], compression techniques (or, compositional reachability analysis) [RGG+95, YY91], symbolic state-space representation [POR12, BCCZ99, BCM+92] and counter-example-guided abstraction refinement (CEGAR) [CGJ+00, CCO+05]. They employ different mechanisms to reduce the state space to be explored or represent it more efficiently. For instance, partial-order reduction allows one to explore only a representative sample of ways in which components of a distributed system can cooperate. Hence, to check whether a property holds, fewer possibilities of cooperation need to be explored. CEGAR, on the other hand, systematically refines an (abstract) over-approximated version of the system until it converges to a representation that is faithful enough to check the desired property. All these techniques have in common their quest for a *precise* reduction/abstraction for the system's state space. Despite being able to tame the state-space explosion problem in many cases, the exact nature of these reductions means they are bound to be as inefficient as simple explicit state-space exploration in other cases; the state-space reduction might not compensate the time taken to carry it out. There is no single method that will work best all the time.

Approximate techniques provide another alternative to deal with the state-space explosion problem [MJ97, AC05, CK94, DCCN04, OPRW13, OCS17, FOSC16, Ant18]. These techniques are built around the fact that a property $\mathcal{P}$ can often be approximated by some *proxy property* $\mathcal{P}'$ satisfying two conditions. If a system satisfies $\mathcal{P}'$, it must also satisfy $\mathcal{P}$, i.e. $\mathcal{P}' \Rightarrow \mathcal{P}$. Also, it must be easier to check $\mathcal{P}'$ than $\mathcal{P}$. The first condition ensures soundness. If such a framework shows that $\mathcal{P}'$ holds, it can soundly deduce that $\mathcal{P}$ holds. Note, however, that the reverse implication ($\mathcal{P} \Rightarrow \mathcal{P}'$) need not hold. By ignoring this implication, we allow approximate frameworks to be incomplete/imprecise. If $\mathcal{P}'$ does not hold for a system, we have no information about $\mathcal{P}$; it might hold or not. This sort of inconclusive result reflects the imprecision of these methods, which is meant to leave some room for efficiency gains. Instead of deciding the original, exact problem, one can look for an incomplete problem with lower complexity. These frameworks purposely sacrifice completeness for efficiency. So, unlike exact methods, they should efficiently analyse all (or most) input systems, albeit, in some cases, imprecisely. In this paper, we introduce a new class of approximate techniques.

A proxy property can be simply created by means of a *reachability over-approximation*. In other words, the approximation is a simpler-to-establish set of states which certainly contains all reachable states but might contain extra ones. Many properties are naturally formulated as (or, can be simply translated into) "no bad state can be reached", where a bad state accounts for some erroneous behaviour. For such properties, replacing exact reachability by some over-approximation creates a proxy property. For instance, deadlock freedom is formulated as "no deadlocked state is in the set of reachable states", and its approximate/proxy counterpart as "no deadlocked state lies within the over-approximation". If this approximation tightly captures the actual state space of a system, it can give rise to a reasonably accurate approximative framework. In a concurrent system, a state is expressed as a tuple containing the state of each component process. Thus, an over-approximation might contain more state tuples than can actually be reached. Obviously, we expect the use of this approximation to speed up the verification process. In this work, we propose a technique to approximate reachability and we analyse the deadlock-freedom and local-deadlock-freedom verification framework it gives rise to, in particular, its precision and scalability. A system is free of local deadlocks if none of its subsystems can become irreversibly blocked. So, for every state, this property has to ensure that *all* (exponentially many) subsystems are not stuck. As this notion of quantification over subsystems is not normally available in traditional verification frameworks, checking this property would typically require a separate state-space exploration for each subsystem. Of course, this sort of analysis would be completely unmanageable.

In this paper, we are particularly interested in how *local analysis* and the invariants they capture can be used to create reachability over-approximations. Verification frameworks normally analyse the global behaviour of the system to yield whether a property holds or not. In many cases, however, a property can be established based on the analysis of small parts of the system. In these cases, verification frameworks can benefit from employing local analysis to examine small subsystems and capture *local invariants*. A local invariant approximates the behaviour of the system based on the behaviour of one of its subsystems; it is meant to show that a system cannot engage in some behaviours because the components in this subsystem cannot cooperate to perform them. This work proposes the notion of *subsystem reachability* as a device to implement local analysis and capture local invariants. It over-approximates system reachability by showing that some system states are not reachable because the components in some subsystem cannot cooperate to reach them.

We also use the notion of subsystem reachability to systematically create a family of reachability over-approximations. For $k \geq 1$, *k-reachability* proposes an approximation that combines reachability for some subsystems of size up to k. This systematisation is an attempt to create an useful reachability approximation that can guide the implementation of a fully automatic verification framework. Without this systematisation, the user of such frameworks would have the unpleasant task of hand-picking some combination of subsystems. Note there are exponentially many combinations to choose from. 1-reachability is a trivial property that simply discards all tuples in which one of the components cannot *by itself* reach the corresponding state. As k increases up to n, the size of the system, k-reachability gets progressively stronger, and obviously equals reachability when k is equal to n.

To demonstrate how local analysis can give rise to useful verification frameworks, we use 2-reachability to create a framework, called *Pair*, that checks deadlock and local-deadlock freedom for concurrent systems. Broadly speaking, *Pair* is the result of replacing exact reachability for 2-reachability. This replacement makes it an approximate framework: *Pair* either shows deadlock (local-deadlock) freedom or it produces an inconclusive result in the sense that it finds a state that is blocked (locally-blocked) and 2-reachable but might not be reachable. This framework's precision is better than current local-analysis-based approximative techniques for deadlock freedom. For example, no other locally-based technique can prove that a system that has some subsystem that *can* deadlock is deadlock (or local-deadlock) free. Such systems are deadlock free but not *hereditarily* deadlock free. This improvement, however, comes with a price. While traditional approximate frameworks are based around polynomially checkable conditions, the problems *Pair* is built around are *NP*-hard. So, we rely on SAT checkers to efficiently implement *Pair* in a way that, typically, scales better than current techniques for deadlock-freedom analysis. These solvers are designed for the purpose of efficiently handling existential quantification. Therefore, they should be particularly effective at tackling the sort of subsystem quantification we use in local-deadlock analysis.

In general, k-reachability gives rise to frameworks that should be more scalable than exact frameworks, but they cannot show properties that depend on some invariant of subsystems of size greater than k. For instance, *Pair* cannot show deadlock (or local-deadlock) freedom if it depends on some local invariant emerging from how triples or larger combinations of components behave. So, to increase *Pair*'s precision, we propose the *PairPicking* framework. It improves 2-reachability by combining *Pair* with subsystem reachability for some subsystems picked by the user. This extension should be useful when proving deadlock (local-deadlock) freedom involves invariants of triples or larger combinations of components, these combinations can be easily identified by the user, and they are much smaller than the entire system. We point out that local analysis alone is not suited to prove properties emerging from the system's *global* behaviour. So, in particular, neither of the present paper's verification frameworks satisfactorily handle such cases. That said, we show that some interaction mechanisms commonly implemented by concurrent systems give rise to local invariants that ensure deadlock and local-deadlock freedom. This work is only concerned with techniques based purely on local analysis. Therefore, we only tangentially discuss techniques that detect global system invariants in Sect. 2. We point out that we have studied a number of ways to effectively combine the sort of local-analysis-based approximations presented in this work with techniques deriving global invariants [AGRR16b, AGRR17a, AGRR17b] and intend to cover these in sequels to this paper.

This paper extends the work in [AGRR16a] as follows.

- We provide a detailed analysis of the deadlock- and local-deadlock-freedom-checking problems and show they are *PSPACE*-complete.
- We generalise the notion of pairwise analysis to that of subsystem reachability, which in turns lead to the generalisation of pairwise reachability into k-reachability.
- We formally analyse the complexity of our reachability approximations and the *Pair* framework, leading to the proof that the problem of finding deadlock candidates for this framework is *NP*-complete, whereas finding local-deadlock candidates in *NP*-hard.
- We introduce the PairPicking strategy; it uses reachability analysis for some selected subsystems (triples or larger combinations of components) to improve *Pair*'s precision.
- We have formally proved our results, most of which were left unproven in the original work.
- We have evaluated our tools with further test cases and against a few other techniques.

Outline Section 2 discusses some work related to ours. In Sect. 3, we introduce the notion of supercombinator machines, upon which this work is based, and we introduce the deadlock- and local-deadlock-freedom-checking problems. Section 4 proposes the local-analysis-based reachability approximations this work revolves around. In particular, we introduce the notions of subsystem and k-reachability to capture and implement local analysis. In Sect. 5, we propose *Pair*, a framework that uses local analysis for proving deadlock and

local-deadlock freedom. Section 6 introduces PairPicking, a strategy that extends *Pair*'s fully-automatic use of local analysis with some manually-provided user inputs. Finally, in Sect. 7, we present our concluding remarks.

2. Related work

Local analysis has been used in many verification frameworks for different contexts and types of concurrency [RD87, BR91, AOS+14, ASW14, OAR+16, AC05, ABB+13, LMC11, Mar96]. [RD87, BR91] introduce a theory of deadlocks based on the notion of an *ungranted request*, that is, a wait-for dependency between components. This work introduces a proof rule based around the *fundamental principle*: under reasonable assumptions about the system, a cycle of ungranted requests is a necessary condition for a deadlock. So, absence of such cycles demonstrates deadlock freedom. This proof rule provides a mathematical tool that can be manually used to show that a system is deadlock free; this work does not propose any verification tool to mechanically support the application of (or, make use of) this rule.

In [AC05, ABB+13, ABB+18, LMC11, Mar96], fully-automated approximate techniques for deadlock freedom are introduced. [AC05] proposes a method for analysing syntactically-restricted shared-variable concurrent programs, whereas the framework in [ABB+13, ABB+18] adapts it to a more general setting meant to describe component-based message-passing systems. [LMC11] proposes a method for architecturally-restricted component-based systems interacting via message passing, and [Mar96] proposes a method for syntactically-restricted message-passing concurrent systems. All these frameworks are based on the fundamental principle (discussed above): they use local analysis to prove the absence of cycles of ungranted requests. From analysing individual and pairs of components, they construct an ungranted-requests digraph and show that such a cycle cannot arise in any conceivable state of the system. These methods tend to be very efficient as this digraph can be constructed and analysed in polynomial time. The use of cycles of dependencies to approximate deadlocks, however, can be imprecise in many ways. Our discussion of SDD (in Sect. 5.1), which is the archetypical framework in this category, clearly exposes these limitations. The framework that we propose in this paper addresses some of these limitations as it relies on a tighter characterisation of (local-)deadlocks and a more precise approach to analyse reachability.

A semi-automatic framework to systematically design deadlock-free systems is introduced in [OAR+16]. It proposes a set of composition rules that only allow components to be composed if they interact properly. Moreover, it introduces a set of refinement expressions that can automatically check whether components interact properly. This framework, however, cannot efficiently tackle systems that have a cyclic communication topology. To cope with that, [AOS+14, ASW14] introduce a set of design patterns that can be used to construct arbitrary-topology systems. The shortcoming of this approach is that there is only a handful of patterns that can be used to construct systems. It also provides a set of refinement expressions to automatically check that a system implements a given pattern. An important benefit of such framework is their guidance in how to construct a deadlock-free system through the use of patterns. The framework that we propose here applies to systems regardless of whether they conform to a pattern or not but they do not provide any clear guide as to how to create a deadlock-free system. For our framework, this burden rests upon the system designer.

Lazy reachability is an exact approach that is based on local analysis [JL16]. It begins analysing the behaviour of a small set of components, and this set is incrementally augmented until either the property is shown or a counter example is found. This approach works well when the property being tested can be demonstrated using local analysis. Otherwise, it will analyse the entire system and thus run into state-space explosion problem. The framework we propose is based on the analysis of small subsystems of a fixed size. Hence, it will not, at any point, analyse the system as whole, which makes the framework efficient but it can also lead to imprecision. In some cases, a property might emerge from the behaviour of a subsystem larger than the subsystems we were set to analyse. In these cases, our framework will not be able to show the validity of the property for the system at hand.

Pure local-analysis techniques cannot prove a property that depends on some global invariant of the system. To circumvent this issue, many frameworks rely on some technique to carry out approximate global analysis [Mar96, CA95, ABC+91, Lam77, AFDR80, BL99, BGL+11, BBL+16, AGRR16b, AGRR17a]. Using additional reachability information [Mar96], data-flow analysis [CK94, DCCN04], or rules to calculate system invariants [Lam77, AFDR80, BL99, BBL+16] are examples of approaches that have been used to implement global analysis. Note that the work presented here is not concerned with global analysis in any way. Instead, it tries to demonstrate the positive impact local analysis alone can have in scaling up verification and to

provide a better outline for the class of properties and systems that can be handled by local analysis and invariants. That being said, in previous work we have examined the combination of pairwise analysis with global invariants and the benefits this might bring to verification frameworks [AGRR16b, AGRR17a].

There exist approximate frameworks that are not based on approximating reachability but in ad-hoc necessary conditions of a given property. For instance, the frameworks in [OPRW13, FOSC16] check livelock freedom whereas the one in [OCS17] checks determinism. The use of conditions that are specific to a property makes these frameworks difficult to adapt for further properties. Conversely, we show in [AGRR17b] how approaches based on replacing exact reachability by approximations, such the one presented here, can be easily adapted to verify properties other than deadlock.

In [Cor96], a number of approaches to tackle the states explosion are discussed, amongst which are compositional techniques, data-flow analysis, symbolic model checking and integer programming. The framework and techniques that we present in this paper are somehow related to these four approaches. Our framework relies on the analysis of small parts of the system, hence it is, to some extent, compositional. These analyses involve explicitly examining the interaction between components, which could be viewed as a type of data-flow analysis. We use these analyses to create a constraint that looks for system states that fulfil a necessary condition for being a deadlocked state; we call such states deadlock candidates. Although expressed in a different language—we propose a boolean constraint as opposed to a system of linear equations with integer solutions—, we are trying to achieve the same purpose as integer-programming-based frameworks, namely, approximate whether a state is *bad* (i.e. deadlocked) by some necessary conditions it must fulfil in order to be so. Finally, we use symbolic techniques, as in SAT solving, to solve this constraint and look for these deadlock candidates. Their absence proves deadlock freedom whereas their presence leads to an inconclusive result: this candidate might or might not be a deadlock. Unlike traditional symbolic model checking, however, where the precise state space is represented by a symbolic constraint, we use a symbolic constraint to deliberately capture an over-approximation of the system state space, which should make the verification of concurrent systems more efficient. That work also evaluates three deadlock-checking frameworks, one of which is an "integer necessary conditions" framework: an approximate approach that shares similarities with ours. Its evaluation shows that it performs well on systems with small components and it is not affected by the communication structures of the system. In our experimentations, we have found similar evidence regarding how systems with small components are more effectively tackled by our framework. However, our approach is affected by the communication topology of the system. The more connections the more analyses of interacting components our framework will need to do.

In [POR12] and [TGS17], two frameworks for symbolic model checking of concurrent systems are proposed. [POR12] proposes an approach to perform refinement checking in a symbolic way. It uses a watchdog approach to create a modified system that reaches an error states if and only if the refinement does not hold. Hence, it transforms refinement checking into a reachability property. Then, it proposes a technique to encode the state space of a system into a boolean formula that can be used for bounded model checking. This framework is later extended to check for an induction step that makes their model-checking framework unbounded. In some cases, this framework outperforms traditional precise explicit-exploration techniques for finding refinement violations. The authors of [POR12], however, admit that their framework is usually much less efficient than traditional explicit-state-exploration techniques for proving that a refinement assertion holds. The authors of [TGS17] propose a boolean encoding for the state space of a system in addition to a set of heuristics that speed up the process of finding error states. They use this approach to create a bounded model checker, which, in particular, cannot guarantee the absence of error states but that they do not exist up to a certain bound i.e. number of transitions. Their experiments suggest that their heuristics outperform general-purpose SAT verification in proving that some state is reachable. The authors do remark, however, that they seem not to help in proving unreachability, even if it is only bounded unreachability. Hence, their framework is useful for finding bugs but not so much for proving safety properties. Unlike both of these approaches, the frameworks and techniques we propose are meant to prove safety properties as opposed to finding bugs. We propose techniques that over-approximate the state space of a system. Therefore, we can prove (unbounded) unreachability by showing that a state does not lie in this approximation, but we do not prove that a given bad state is reachable. If a state lies in the approximation, we do not know whether it truly belongs to the state space of the system or whether it does not. Thus, our techniques should complement the above ones.

3. Background

In this section, we introduce *supercombinator machines*, the notation upon which our work is based. This notation is used by FDR4 [GRABR14] to capture and implement CSP systems [Hoa85, Ros10]. Communicating Sequential Processes (CSP) [Hoa85, Ros10] is a notation used to model concurrent systems where processes interact by exchanging messages. 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 [Ros10]. Supercombinator machines will capture the semantics of concurrent systems in virtually all process algebras. In particular, they can support a wide variety of schemes involving hiding of communications and renaming.

FDR4 captures components of a concurrent systems using *labelled transition systems*. We use $\mathcal{E}$ to denote the finite universal set of *visible* events, $\tau \notin \mathcal{E}$ the *invisible* event, and $\checkmark \in \mathcal{E}$ the termination signal.

Definition 1 A labelled transition system (LTS) is a 4-tuple $(S, \Sigma, \Delta, \hat{s})$ where S is a non-empty set of states, $\Sigma \subseteq \mathcal{E} \cup \{\tau\}$ is the alphabet, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation, and $\hat{s} \in S$ is the starting state.

We use $s \xrightarrow{a} s'$ if and only if $(s, a, s') \in \Delta$, $s \xrightarrow{a}$ if and only if $\exists\, s' \in S \bullet s \xrightarrow{a} s'$, and $s \xrightarrow{\langle a_1, \ldots, a_n \rangle} s'$ denotes the existence of a path from s to s' with a sequence of events $\langle a_1, \ldots, a_n \rangle$, namely, there exist $s_0, \ldots, s_n$ such that for all $i \in \{0 \ldots n-1\}$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$, and $s_0 = s$ and $s_n = s'$.

Instead of using the SOS (Structured Operational Semantics) rules to explicitly generate the LTS of a system [Plo81], FDR4 relies on a *combinator-based* operational semantics (see [Ros10]) that represents systems as supercombinator machines. A supercombinator machine represents a concurrent system by the LTSs of component processes and a set of rules that set out how these components can interact. Any combinator semantics can be translated into a SOS one.

Definition 2 A *single-format 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:
  - $e \in (\mathcal{E} \cup \{\tau, -\})^n$ specifies the event that each component must perform, where $-$ indicates that the component performs no event. At least one component must perform an event.
  - $a \in \mathcal{E} \cup \{\tau\}$ is the event the supercombinator machine performs.

FDR4 works with a version of a supercombinator machine that might have multiple *formats*. Formats are partitions of the machine's rules. For these machines, each rule is associated with a format and rule application triggers a (possible) change of format. In this work, however, we have the restriction that we only deal with single-format machines. Note, however, that a multi-format machine can be translated into a single-format machine with an equivalent behaviour in polynomial time. This translation would involve adding a new component to track the system's current format and modifying the machine's rules to comply with format restrictions and to perform format changes. Nevertheless, we impose this restriction because the techniques we propose should be better suited to handle systems that are naturally described by single-format machines. This artificial translation into a single-format machine is likely to damage the precision of our techniques in capturing the behaviour of the original non-single-format system. In practice, many systems are naturally modelled by single-format machines; systems that are constructed in CSP using the replicated-alphabetised-parallel operator, for instance, are naturally represented in this way.

The frameworks that we propose in this work are intended to be more precise when applied to machines that are not only single-format but also *triple-disjoint*.

Definition 3 A *supercombinator machine* $(\mathcal{L}, \mathcal{R})$ with n components is triple disjoint if and only if for all pairs $(e, a) \in \mathcal{R}$, e is triple disjoint, that is, at most two components participate in a rule.

$$triple\_disjoint(e) = \forall\, i, j, k \in \{1 \ldots n\} \mid i \neq j \land j \neq k \land i \neq k \bullet (e_i = -) \lor (e_j = -) \lor (e_k = -)$$

Triple disjointness is a restriction but a single-format machine can also be translated in polynomial time into a triple-disjoint single-format machine. This translation is more complicated than the previous one and involves the creation of new components, one per rule of the original machine, to emulate rule application. Again, we impose this restriction because we believe the frameworks we propose should, in general, analyse more accurately this type of machines. We also point out that many supercombinator machines are naturally triple disjoint. Moreover, triple disjointness is also a restriction imposed by many notations and frameworks that are

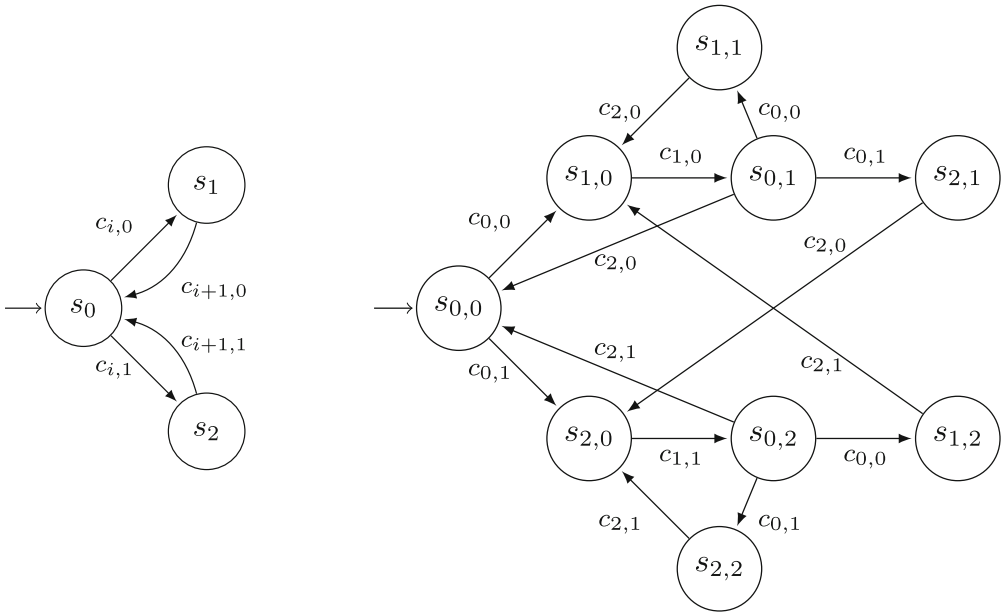

Fig. 1. LTS for components L_i and that of $\mathcal{S}_{MS_2}$

similar to ours [Ram11, Mar96, AC05, FOSC16, AOS+14, ASW14]. Henceforth, we use the term supercombinator machine instead of single-format triple-disjoint supercombinator machine.

We illustrate the notion of a supercombinator machine with an example.

Example 1 We construct a 2-place buffer, which stores boolean values 0 and 1, by putting together two 1-place buffers depicted by components L_0 and L_1 in Fig. 1. In our model, event $c_{i,v}$ ($c_{i+1,v}$) represents the input (output) of value v by component i. The system implementing our 2-place buffer is represented by the supercombinator machine $\mathcal{S}_{MS_2} = \{\langle L_0, L_1\rangle, \mathcal{R}\}$, where $\mathcal{R}$ requires the synchronisation of shared events:

$$\mathcal{R} = \{((c_{0,0}, -), c_{0,0}), ((c_{0,1}, -), c_{0,1}), ((c_{1,0}, c_{1,0}), c_{1,0}), ((c_{1,1}, c_{1,1}), c_{1,1}), ((-, c_{2,0}), c_{2,0}), ((-, c_{2,1}), c_{2,1})\}.$$

A supercombinator machine is an implicit representation of a system in the sense that it *induces* a LTS representing its behaviour. The LTS induced by $\mathcal{S}_{MS_2}$, for instance, is presented in Fig. 1; we use $s_{i,j}$ to denote state (s_i, s_j). Note that this very trivial example already gives rise to a state space that is a quite hard to read and understand. Hence, the need for sound automatic tools for the analysis of concurrent systems.

Definition 4 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:

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

In this work, we use *system state* (*component state*) to designate a state in the system's (component's) LTS. From now on, we refer to a system and its supercombinator machine interchangeably. According to our definition of a system's induced LTS, a state might be reachable or not. We assume, however, that all states in a component LTS are reachable.

Definition 5 For induced LTS $(S, \Sigma, \Delta, \hat{s})$, state $s \in S$ is reachable if and only if $reachable(s)$ holds, where $reachable(s) = \exists\, p \in \Sigma^* \bullet \hat{s} \xrightarrow{p} s$.

The reason for choosing supercombinator machines to reason about concurrent and distributed systems is two-fold. Firstly, these machines are simple and can seamlessly capture the behaviour of systems described in many common formalisms. Even though we rely on CSP to model systems and FDR4 to compile them into supercombinator machines in our implementation later on, our framework should be easily adaptable to similar formalisms; a new compilation procedure to transform systems described in this new formalism into supercombinator machines should be the only requirement for this adaptation. Secondly, this operational notion, as intended, provides a system description that is fairly simple to implement and manipulate when constructing analysis tools.

FDR4 is a fully-automatic verification tool that checks properties about CSP systems. In this work, we are concerned with the problem of verifying *deadlock and local-deadlock freedom*. We introduce these problems, discuss whether/how FDR4 tackles them, and show that they are *PSPACE*-complete. Currently, there is no known algorithm that can solve such problems in polynomial time, and the common belief is that there is none. In fact, it is widely believed that *PSPACE*-complete problems can only be solved in at least exponential time.

To show *PSPACE*-hardness, we rely on reductions from the *single-component traces-refinement problem*. For a specification LTS L_{sp} and an implementation LTS L_I, both of which are finite, this problem asks whether $L_{sp} \sqsubseteq_T L_I$ holds, where $L_{sp} \sqsubseteq_T L_I$ if and only if $traces(L_I) \subseteq traces(L_{sp})$, $traces(L) = \{tr \mid \exists\, s \in S \bullet \hat{s} \overset{tr}{\Longrightarrow} s\}$, and $s \overset{tr}{\Longrightarrow} s'$ holds when tr is a trace leading L from s to s', that is, there exist a path $s \xrightarrow{\langle a_1,\ldots,a_n\rangle} s'$ such that tr is the sequence of events resulting from removing all τ-occurrences from $\langle a_1, \ldots, a_n\rangle$. Intuitively, an implementation L_I refines a specification L_{sp} in the traces model, namely, $L_{sp} \sqsubseteq_T L_I$ holds, if the implementation can only perform behaviours (i.e. traces) that are allowed (i.e. are performed) by the specification. This is why the traces of specification and implementation are related in the opposite way: $traces(L_{sp}) \supseteq traces(L_I)$. This problem has been shown to be *PSPACE*-complete in [KS90]:

Lemma 1 Given a specification LTS L_{sp} and an implementation LTS L_I, the problem of checking that $L_{sp} \sqsubseteq_T L_I$ holds is *PSPACE*-complete.

Moreover, we also use the following lemma, a direct consequence of *Savich's theorem* [Sav70], in proving *PSPACE* membership. Savich's theorem provides a construction to translate a non-deterministic algorithm that uses polynomial space into a deterministic procedure with a quadratic space increase.

Lemma 2 *NPSPACE = PSPACE.*

For the sake of decidability, we only consider supercombinator machines with a finite number of components, which are themselves represented by finite LTSs. Intuitively, the need to explore induced LTSs and their state-space explosion can be seen as the cause for this problem's membership to *PSPACE*. We use the following definitions for the size of a supercombinator machine and a LTS.

Definition 6 Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS and $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ a supercombinator machine:

- the size of $\mathcal{S}$ is given by $|\mathcal{S}| = (\sum_{i\in\{1\ldots n\}} |L_i|) + (n \cdot |\mathcal{R}|)$;
- the size of L is given by $|L| = |S| + |\Delta|$.

A system deadlocks when it reaches a state in which it becomes *blocked*, namely, unable to perform any further event. So, a system is deadlock free if no such state exists.

Definition 7 Given a supercombinator machine $\mathcal{S}$, the deadlock-freedom problem asks whether $\mathcal{S}$'s induced LTS $L = (S, \Sigma, \Delta, \hat{s})$ is deadlock free, namely, whether $\neg\, \exists\, s \in S \bullet deadlock(s)$ holds.

- $deadlock(s) = reachable(s) \wedge blocked(s)$
- $blocked(s) = \neg(s \longrightarrow)$, where $s \longrightarrow$ holds if and only if $\exists\, e \in \Sigma \bullet s \xrightarrow{e}$

Deadlock-freedom is a very important property in practice. Checking this property is often considered the first step towards showing that a concurrent/distributed system is correct. Moreover, many safety properties can be reduced to verifying deadlock freedom of modified systems [GW93].

This problem's *PSPACE*-completeness means that automatic verification techniques usually struggle to show deadlock freedom even for systems with a rather small number of components. FDR4 has a built-in assertion that checks deadlock freedom. It implements a *breadth-first-search* algorithm to explicitly explore the induced LTS of a system, looking for a deadlock.

Theorem 1 The deadlock-freedom-checking problem is *PSPACE*-complete.

Proof We prove that the deadlock-freedom-checking problem is (i) in *PSPACE* and (ii) *PSPACE*-hard.

Firstly, we show (i) by providing a high-level description for a non-deterministic Turing machine (NTM) that uses polynomial space to check deadlock freedom.

Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be the supercombinator machine we try to show to be deadlock free, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $L = (S, \Sigma, \Delta, \hat{s})$ is its induced LTS. We call NTM CHECK in Algorithm 1 with input $(\mathcal{S}, \hat{s}, |S|)$, where $|S| = \prod_{i \in \{1 \ldots n\}} |S_i|$. If it accepts this input, $\mathcal{S}$ has a deadlock, if it rejects, $\mathcal{S}$ is deadlock free. This machine accepts some input if some branch yields *accept*, and rejects it if all branches yield *reject*. Each *guess* creates a branch that has to store the supercombinator $\mathcal{S}$, a state s, a number n, and some extra space to calculate $blocked(s)$ and $sucessor(s)$, namely, the memory used is proportional to the size of $\mathcal{S}$. So, it uses polynomial space.

Moreover, it correctly decides deadlock-freedom. If a deadlock exists, it must be at most $|S|$ transitions away from $\hat{s}$, since there must be a simple path in L leading to it, and our procedure goes through all paths of length at most $|S|$.

Algorithm 1 NTM to check for a deadlock for machine $\mathcal{S}$.

```
1: function CHECK(S, s, n)
2:     if n > 0 then
3:         if blocked(s) then accept
4:         else guess s' ∈ successors(s) :
5:             CHECK(S, s', n − 1)
6:         end if
7:     else reject
8:     end if
9: end function
```

Secondly, we show (ii) by providing a polynomial-time reduction from the single-component traces-refinement problem to the deadlock-freedom checking problem.

Let $L_{sp} = (S_{sp}, \Sigma_{sp}, \Delta_{sp}, \hat{s}_{ap})$, where $S_{sp} = \{s_1, \ldots, s_n\}$, be a specification LTS and L_I an implementation LTS with alphabet Σ_I. We assume, without loss of generality, that L_{sp} and L_I are τ-free and $\tau \notin \Sigma_I \cup \Sigma_{sp}$; we could run a τ-elimination procedure that runs in polynomial time on L_{sp} and L_I and creates a traces-equivalent τ-free LTS. We propose the creation of the supercombinator machine $\mathcal{T}(L_{sp}, L_I)$, described next, as a means of reducing the verification of single-component traces refinement between these LTSs to checking deadlock freedom for this machine. Components S_i, C_i and CC are described below.

$$\mathcal{T}(L_{sp}, L_I) = (\langle S_1, C_1, \ldots, S_n, C_n, CC, L_I \rangle, \mathcal{R})$$

Much like checking language containment for non-deterministic automata, traces refinement is usually carried out by exploring the product space of the *determinised* specification and the implementation. This product space matches pairs of states that can be reached via the same trace and the exploration checks whether the implementation state can perform a subset of the events the specification state can. The reason for determinising the specification is to avoid having to compare an implementation state with (possibly) multiple specification states; non-determinism causes a LTS to reach two different states with the same trace. The determinisation procedure creates states that correspond to sets of states of the original specification.

In this reduction, we cannot afford determinising the specification upfront, as this would lead to an exponential time reduction. Instead, we create a supercombinator machine that can be understood as behaving like the same specification-implementation product LTS but it carries out a sort of lazy determinisation.

The machine $\mathcal{T}(L_{sp}, L_I)$ runs components $\langle S_1, C_1, \ldots, S_n, C_n, CC, L_I \rangle$ in parallel. The components S_1, $C_1, \ldots, S_n, C_n$ account for the determinised behaviour of the specification, L_I is the implementation component, and CC is a central controller that ensures specification and implementation reach trace-matching states. Furthermore, CC also goes into a deadlock state, causing the entire machine to deadlock, if a pair of *violating* states is found, namely, a pair of specification and implementation states where the implementation can perform an event not allowed by the specification.

Component CC, roughly speaking, reads an event the currently-being-visited implementation state can perform and tries to see if the determinised specification counterpart state can perform it too. If so, it moves

on to next pair of states to be visited. On the other hand, if the specification cannot perform this event it goes into a deadlock state, which leads the entire system into a deadlock. In this machine, in addition to the original events e, we use fresh events e' (and $\bar{e}'$) to denote that the determinised specification can (resp. cannot) give rise to event e, respectively. CC definition is given next, and we provide a sketch of its graphical representation in Fig. 2.

$CC = (S, \Sigma, \Delta, \hat{s})$ where

- $S = \{start, good, bad, idle\} \cup \{s_e \mid e \in \Sigma_I\}$
- $\Sigma = \{ready, reset\} \cup \{e, \bar{e}', e' \mid e \in \Sigma_I\}$
- $\Delta = \{(start, e, s_e) \mid e \in \Sigma_I)\} \cup \{(start, \tau, start)\}$
  $\cup\{(s_e, e', good), (s_e, \bar{e}', bad) \mid e \in \Sigma_I)\}$
  $\cup\{(good, reset, idle), (idle, ready, start)\}$
- $\hat{s} = start$

We use components $S_1, \ldots, S_n$ to represent a state of the determinised specification. Component S_i accounts for the behaviour of state s_i in L_{sp}. S_i's states *on* and *off* tell whether s_i is part of the determinised state currently being visited, whereas states s_e and s'_e serve to update what the next determinised state should be based on the successors of s_i for event-e transitions in L_{sp}. We use the fresh event suc_e to carry out this update as we explain later. We provide a definition for this component next, and a sketch of its graphical representation in Fig. 2.

$S_i = (S, \Sigma, \Delta, \hat{s})$ where

- $S = \{on, off\} \cup \{s_e \mid s_i \xrightarrow{e}\}$
- $\Sigma = \{ready, on_i\} \cup \{suc_e, e', \bar{e}' \mid s_i \xrightarrow{e}\}$
- $\Delta = \{(off, on_i, on), (off, ready, off)\} \cup \{(off, e', off), (off, \bar{e}', off) \mid s_i \xrightarrow{e}\}$
  $\cup\{(on, on_i, on), (on, ready, on)\} \cup \{(on, e', s_e) \mid s_i \xrightarrow{e}\}$
  $\cup\{(s_e, reset, s'_e), (s'_e, suc_e, off) \mid s_i \xrightarrow{e}\}$
- $\hat{s} = on$ if $\hat{s}_{sp} = s_i$, and $\hat{s} = off$, otherwise

Component C_i is a simple controller to S_i. It keeps information about whether this state should be part of the next determinised state being visited, namely, whether S_i should be "turned on". The definition for this component is given next. Also, we provide its graphical representation in Fig. 2.

$C_i = (S, \Sigma, \Delta, \hat{s})$ where

- $S = \{on, off\}$
- $\Sigma = \{ready, on_i, start_i\}$
- $\Delta = \{(off, start_i, on), (off, ready, off)\}$
  $\cup\{(on, on_i, off), (on, start_i, on)\}$
- $\hat{s} = off$

Finally, we describe, as follows, the rules that regulate the interaction between components in this machine. We use the set $\{(i_1, a_1), \ldots, (i_m, a_m)\}$ as an alternative way to represent the tuple $(e_1, \ldots, e_n)$ where $e_{i_j} = a_j$ if the pair (i_j, a_j) belongs to the set and $-$ otherwise. Moreover, we use the name of components to represent their position in a rule's event tuple: $S_i = 2i - 1$, $C_i = 2i$, $CC = 2n + 1$, $L_I = 2n + 2$. For instance, the rule $(\{(i, ready) \mid i \in \{1 \ldots 2n + 1\}\}, ready)$ requires all components of $\mathcal{T}(L_{sp}, L_I)$ except for L_I to synchronise on event $ready$ leading to system event $ready$, whereas the rule $(\{(S_i, on_i), (C_i, on_i)\}, on_i)$ requires components S_i and C_i to synchronise on on_i to produce the system event on_i.

$$
\begin{aligned}
\mathcal{R} = &\{(\{(i, ready) \mid i \in \{1 \ldots 2n + 1\}\}, ready), (\{(CC, \tau)\}, \tau)\} \\
&\cup \{(\{(S_i, on_i), (C_i, on_i)\}, on_i) \mid i \in \{1 \ldots n\}\} \\
&\cup \{(\{(CC, e), (L_I, e)\}, e) \mid e \in \Sigma_I\} \\
&\cup \{(\{(S_i, reset) \mid i \in \{1 \ldots n\}\} \cup \{(CC, reset)\}, reset)\} \\
&\cup \{(\{(S_i, e') \mid i \in \{1 \ldots n\} \wedge s_i \xrightarrow{e}\} \cup \{(CC, e')\}, e') \mid e \in \Sigma_I\} \\
&\cup \{(\{(S_i, \bar{e}') \mid i \in \{1 \ldots n\} \wedge s_i \xrightarrow{e}\} \cup \{(CC, \bar{e}')\}, \bar{e}') \mid e \in \Sigma_I\} \\
&\cup \{(\{(C_j, start) \mid j \in \{1 \ldots n\} \wedge s_i \xrightarrow{e} s_j\} \cup \{(S_i, suc_e)\}, suc_e) \mid i \in \{1 \ldots n\} \wedge e \in \Sigma_{sp}\}
\end{aligned}
$$

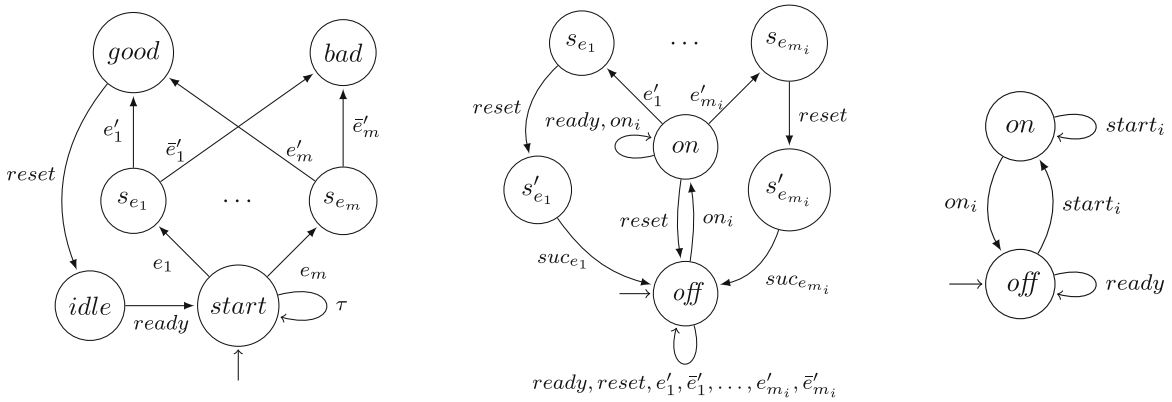

Fig. 2. LTS sketches of component CC for $\Sigma_I = \{e_1, \ldots, e_m\}$ and of component S_i with $\{e \mid s_i \xrightarrow{e}\} = \{e_1 \ldots e_{m_i}\}$, and LTS of component C_i, respectively

All but the last set in this definition require components to synchronise in shared events. The last set, however, triggers the creation of the next determinised specification state to be visited. Given the event performed by the implementation, our machine has to update the determinised specification state accordingly. Assuming that e was performed by the implementation, this update involves "activating", for each state s_i (i.e. component S_i) active in the current determinised state, all successor state s_j (i.e. component S_j) such that $s_i \xrightarrow{e} s_j$ in L_{sp}. The rules in this last set mimic this behaviour; component S_i performing suc_e triggers $start_j$ to be performed by all C_j associated with s_i's successors.

To conclude our proof, we show that (iii) this machine is deadlock-free *iff* $L_{sp} \sqsubseteq_T L_I$ holds and that (iv) it can be constructed in polynomial time.

As our machine is built to mirror the behaviour of the product space of the determinised specification and implementation, (iii) holds. Note that CC only reaches state *bad iff* $L_{sp} \sqsubseteq_T L_I$ does not hold, as per Lemma 3, and making CC reach *bad* is the only way our machine can deadlock, as per Lemma 4. By the definitions of the components in this machine and its rules, it should be clear that (iv) holds. Each component C_i can be constructed in constant time, components $S_1, \ldots, S_n$ can be constructed in time proportional to $|L_{sp}|$, and CC can be constructed in time proportional to $|L_I|$. QED □

Lemma 3 CC reaches state *bad* within $\mathcal{T}(L_{sp}, L_I)$ *iff* $L_{sp} \sqsubseteq_T L_I$ does not hold.

Proof The system $\mathcal{T}(L_{sp}, L_I)$ mirrors the behaviour of a refinement checking precedure for specification L_{sp} and implementation L_I. A refinement checking procedure would, first, create L_{det} a determinised version of L_{sp}. In this determinised version, no two different states can be reached via the same trace and determinised states correspond to set of states of the original specification. For instance, if states s_1 and s_2 are reached (only) via tr in L_{sp}, the state $\{s_1, s_2\}$ must be reached via tr in L_{det}. A refinement checking procedure explores the product LTS $L_{I \times det}$ of L_I and L_{det}. Let (s_I, s_{det}) be a state of $L_{I \times det}$ (i.e. in the product space of L_I and L_{det}) that is reached via trace tr , the procedure checks if there is an event $e \in initials_{L_I}(s_I)$ that is not in any $initials_{L_{sp}}(s_j)$ where $s_j \in s_{det}$. If there is such an event, it reports that the trace $tr\hat{}\langle e \rangle$ is a traces-refinement violation. Otherwise, it explores the successors of state (s_I, s_{det}) in $L_{I \times det}$ reached via an e-transition. A transition $(s_I, s_{det}) \xrightarrow{e} (s'_I, s'_{det})$ exists in $L_{I \times det}$ iff $s_I \xrightarrow{e} s'_I$ and $s'_{det} = \{s' \mid s \in s_{det} \wedge s \xrightarrow{e} s' \text{ in } L_{sp}\}$; this transition definition matches a state s_I reached in L_I via tr with the set of states reached in L_{sp} via the same trace tr. The initial state of $L_{I \times det}$ is $(\hat{s}_I, \{\hat{s}_{sp}\})$ where $\hat{s}_I$ and $\hat{s}_{sp}$ are the initial states of L_I and L_{sp}, respectively. Therefore, the refinement-checking exploration starts in this state.

Our system mirrors exactly this behaviour by using components $S_1, \ldots, S_n$ to capture determinised states, and component CC to guide the search and check pairs of states in this product space. Let us call the states of $\mathcal{T}(L_{sp}, L_I)$ where CC is in *start*, *checkpoint* states. Moreover, for a state s of $\mathcal{T}(L_{sp}, L_I)$ and $\{s_1, \ldots, s_n\}$ the states of L_{sp}, we use $det(s)$ to represent the set $\{s_i \mid s_i \in \{s_1, \ldots, s_n\} \wedge S_i \text{ is in state } on \text{ in } s\}$; the function det outputs the determinised state given by which components S_i are on. Also, we use $I(s)$ to give the state of component L_I in the state s of $\mathcal{T}(L_{sp}, L_I)$. The following mirroring holds:

i. $(s_I, s_{det}) \xrightarrow{e} (s'_I, s'_{det})$ in $L_{I\times det}$ if and only there is a path $s \xrightarrow{tr} s'$ in the LTS induced by $\mathcal{T}(L_{sp}, L_I)$ such that $tr \restriction \Sigma_I = \langle e \rangle$, s and s' are checkpoint states, $s_{det} = det(s)$, $s_I = I(s)$, $s'_{det} = det(s')$, and $s'_I = I(s')$. While Σ_I gives the alphabet of L_I, $tr \restriction \Sigma_I$ gives rise to the trace resulting from removing all occurrences of events not in Σ_I from tr.

First we show the direction ($\Rightarrow$); we assume that (a) $(s_I, s_{det}) \xrightarrow{e} (s'_I, s'_{det})$ is a transition of $L_{I\times det}$ and that (b) s is a state of $\mathcal{T}(L_{sp}, L_I)$ such that s is a checkpoint state, $s_{det} = det(s)$ and $s_I = I(s)$, and show that there is path engaging on a trace tr and leading to a state s' such that s' is a checkpoint state, $tr \restriction \Sigma_I = \langle e \rangle$, $s'_{det} = det(s')$, and $s'_I = I(s')$. From (b), we must have that in state s, CC is in $start$, all $C_1, \ldots, C_n$ are in *off*, L_i is in s_I and the components S_i that are turned on capture the determinised state s_{det}. From (a), we know that (c) L_I has an e-transition from s_I to s'_I and, hence, L_I and CC can synchronise on e, leading CC to s_e and L_I to s'_I. At this point, all S_i components that can perform an e' transition can synchronise with CC. This leads CC to $good$, the S_i components that can engage on e' and are turned on move to their state $s_{e'}$ whereas the ones that are turned off remain so. Note that there might be S_i components that are on and cannot engage on e'; these components remain on after this synchronisation. Then, all S_i components synchronise with CC on reset. This move CC to idle and the components S_i in s_e to s'_e, whereas the S_i components in on or *off* move to *off*. Then, each component S_i that is in s'_e is ready to activate the components C_j for which $s_i \xrightarrow{e} s_j$ in L_{sp}; this occurs thanks to the synchronisation between suc_e for S_i and all $start_j$ for components C_j. While the synchronisations on suc_e turn off the remaining S_i components still not in *off*, the synchronisations on on_j for C_j and S_j (leading C_j to *off* and S_j to on) re-activate the components S_j that will form the next determinised state. Once all such synchronisations occur, the next determinised state has been set and components $CC, S_1, C_1, \ldots, S_n, C_n$ are ready to synchronise on $ready$, which makes CC move to state $start$ and so a new checkpoint state s' is reached. Given that events suc_e activate the right successors S_j for S_i, we have that $s'_{det} = det(s')$, also $s'_I = I(s')$ as per (c), and since e was the only event in Σ_I that the system engaged on to reach s', we have that $tr \restriction \Sigma_I = \langle e \rangle$.

A very similar argument can be used to show the opposite direction, namely, that given a path $s \xrightarrow{tr} s'$ in the LTS induced by $\mathcal{T}(L_{sp}, L_I)$ such that $tr \restriction \Sigma_I = \langle e \rangle$, s and s' are checkpoint states, and a state (s_I, s_{det}) of $L_{I\times det}$ such that $s_{det} = det(s)$ and $s_I = I(s)$, there must be a transition $(s_I, s_{det}) \xrightarrow{e} (s'_I, s'_{det})$ in $L_{I\times det}$ where $s'_{det} = det(s')$, and $s'_I = I(s')$.

Now we can prove our theorem using (**i**). Firstly, we show that if $\mathcal{T}(L_{sp}, L_I)$ reaches a state s'' where CC is in bad then $L_{sp} \sqsubseteq_T L_I$ does not hold. To reach such a state, $\mathcal{T}(L_{sp}, L_I)$ has to engage on a path with trace $tr\hat{}\langle e, \bar{e}' \rangle$. The trace tr leads the system to a checkpoint state, from which it engages on an event e and then an event $\bar{e}'$. Since $\mathcal{T}(L_{sp}, L_I)$ and $L_{I\times det}$ start from states satisfying the condition in (**i**), after tr and $tr \restriction \Sigma_I$ they reach mirrored states s' and (s'_I, s'_{det}), respectively, such that $det(s') = s'_{det}$ and $I(s') = s'_I$. From checkpoint state s' to state s'', $\mathcal{T}(L_{sp}, L_I)$ engages on e (synchronisation of L_I and CC in e) and then $\bar{e}'$ (synchronisation of all S_i that can perform e' but are in *off* and CC in $\bar{e}'$). This means that the implementation component L_I can perform e but all components S_i that can perform e' are *off*. As the components S_i mirror determinised states of L_{det}, it means that after $tr \restriction \Sigma_I$, L_I can reach a state where it can perform e whereas L_{det} reaches a state in which no specification state can perform e. Therefore, $(tr \restriction \Sigma_I)\hat{}\langle e \rangle$ represents a violation for $L_{sp} \sqsubseteq_T L_I$.

We can also use (**i**) to prove the opposite direction of our theorem, namely, if $L_{sp} \sqsubseteq_T L_I$ does not hold then $\mathcal{T}(L_{sp}, L_I)$ reaches a state s'' where CC is in bad. Let us assume without loss of generality that $tr\hat{}\langle e \rangle$ is a violation to $L_{sp} \sqsubseteq_T L_I$, where tr is a trace that L_{sp} can perform. We can use (**i**) to show that there is a trace tr' where $tr' \restriction \Sigma_I = tr$, and such that tr' and tr lead $\mathcal{T}(L_{sp}, L_I)$ and $L_{I\times det}$ to mirrored states s' and (s'_I, s'_{det}), respectively. Since e leads to a violation of $L_{sp} \sqsubseteq_T L_I$, it must be that after tr, $L_{I\times det}$ reach state (s'_I, s'_{det}) such that L_I can engage on e from s'_I and L_{sp} cannot engage on e from any of the specification states in s'_{det}. Hence, as per the definition of $\mathcal{T}(L_{sp}, L_I)$, from s', the component L_I can synchronise with CC in event e, and then, as all components S_i that can perform e' must be *off*, they synchronise with CC in event $\bar{e}'$. This leads the system to state s''. QED □

Lemma 4 $\mathcal{T}(L_{sp}, L_I)$ only deadlocks when CC is in state bad.

Proof As we have explained in the proof of the previous lemma, the components in $\mathcal{T}(L_{sp}, L_I)$ behave in a way that $S_1, C_1, \ldots, S_n, C_n$ and L_I can always synchronise with CC and make the system progress. L_I might refuse to synchronise with CC if it reaches a sink state (i.e. without outgoing transitions). This can only occur, however, when CC is in $start$. Note that in this state CC has a τ self loop which makes CC

advance individually and the system progress. All the other components (S_1, C_1, ..., S_n, C_n) do not have sink states, hence they can always synchronise with CC. Nevertheless, CC might transition to bad, at which point, it refuses to synchronise with all other components and blocks the entire system. QED □

Local-deadlock freedom is another property that we are interested in. We want to show that no subsystem can become irreversibly blocked. In many cases, instead of deadlock freedom, system designers are actually interested in achieving local-deadlock freedom. As opposed to creating systems where a single component makes the system progress while others are forever stuck, it seems more reasonable to have *all* components effectively interacting and contributing to the overall behaviour of the system. Deadlock freedom is often checked of concurrent systems as a way to check for basic design flaws. Local deadlock freedom is a more discerning way of doing so. Note that while local-deadlock freedom ensures that each (subsystem) state has some outgoing transition, starvation freedom requires components to eventually participate on a possible system transition, namely, a component cannot be prevented from engaging on some system transition because of some poor scheduling that favours other components in spite of this one [BK91, HS08]. The techniques that we propose in this work are meant to tackle safety properties, namely, they can show that a bad state cannot be reached. Therefore, they can prove deadlock and local-deadlock freedom. On the other hand, they were not designed to and cannot generally prove progress/liveness properties such as starvation freedom.

We determine whether a subsystem ss is irretrievably blocked by examining transitions derived from the projected set of system rules $\mathcal{R}_{ss}$. It projects the rules in $\mathcal{R}$ which require the participation of some component $i \in ss$ (predicate on captures that) in a way that the projected rule $r \restriction ss$ disregards (does not require) the participation of system components not in ss.

Definition 8 Given a supercombinator machine $\mathcal{S} = (\mathcal{L}, \mathcal{R})$ where $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$, the local-deadlock-freedom problem asks whether there exists a system state s such that $\mathit{local\text{-}deadlock}(s)$ holds.

- $\mathit{local\text{-}deadlock}(s) = \mathit{reachable}(s) \wedge \mathit{locally\text{-}blocked}(s)$
- $\mathit{locally\text{-}blocked}(s) = \exists\, ss \in \mathbb{P}(\{1 \ldots n\}) \mid ss \neq \emptyset \bullet \mathit{blocked}_{ss}(s)$
- $\mathit{blocked}_{ss}(s) = \neg\, s \xrightarrow{\mathcal{R}_{ss}}$, where $s \xrightarrow{\mathcal{R}_{ss}}$ is the predicate $s \longrightarrow$ as per Definition 7 for the LTS induced by $(\mathcal{L}, \mathcal{R}_{ss})$.
  - $R_{ss} = \{r \restriction ss \mid r \in \mathcal{R} \wedge \mathit{on}(r, ss)\}$.

  For the following two definitions, let $r = ((e_1, \ldots, e_n), a)$.
  - $\mathit{on}(r, ss) = \exists\, i \in ss \bullet e_i \neq -$
  - $r \restriction ss$ gives rise to tuple $((e'_1, \ldots, e'_n), a)$ where $e'_i = e_i$ if $i \in ss$ and $e'_i = -$ otherwise.

Note that this property establishes that all (exponentially many) subsystems are not blocked. As most traditional verification frameworks do not explicitly handle this sort of quantification, one normally has to explicitly devise exponentially many separate checks to analyse all subsystems. Moreover, local-deadlock freedom implies deadlock freedom as the entire system is indeed one of the analysed subsystems. Of course, the converse does always not hold.

Lemma 5 A local-deadlock-free system must also be deadlock free.

The problem of deciding whether a system is free of local deadlocks is also *PSPACE*-complete. Note that FDR4 does not have a built-in assertion for local-deadlock freedom and to do so would seemingly require a separate provision for each subsystem.

Theorem 2 The local-deadlock-freedom-checking problem is *PSPACE*-complete.

Proof We prove that the local-deadlock-freedom-checking problem is (i) in *PSPACE* and (ii) *PSPACE*-hard.

Firstly, we show (i) using NTM CHECK' in Algorithm 2, which checks local-deadlock freedom using polynomial space.

Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be the supercombinator machine we trying to show to be free of local deadlocks, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $L = (S, \Sigma, \Delta, \hat{s})$ is its induced LTS. We call NTM CHECK' in Algorithm 2 with input $(\mathcal{S}, \hat{s}, |S|)$. If it accepts this input, $\mathcal{S}$ has a local deadlock, if it rejects, $\mathcal{S}$ is deadlock free. Each branch that has to store the supercombinator $\mathcal{S}$, a state s, a number n, some space to compute $\mathit{blocked}_{ss}(s)$ and $\mathit{sucessor}(s)$. To compute $\mathit{blocked}_{ss}(s)$, we create and store the projected rules $\mathcal{R}_{ss}$. Note that this set

is, at most, as big as $\mathcal{R}$. Also, obviously, we reuse memory between $blocked_{ss}(s)$ computations. So, the space used by this machine is proportional to (and polynomial in) the size of $\mathcal{S}$.

Moreover, it correctly decides local-deadlock freedom, as a local-deadlock must be reachable by a simple path of L of length at most $|S|$, and our machine examines all such paths.

Algorithm 2 NTM to check for a local deadlock for machine $\mathcal{S}$.

```
1:  function CHECK'(S, s, n)
2:      if n > 0 then
3:          for all ss ⊆ {1 ... n} | ss ≠ ∅ do
4:              if blocked_ss(s) then accept
5:              end if
6:          end for
7:          guess s' ∈ successors(s) :
8:              CHECK'(S, s', n − 1)
9:      else reject
10:     end if
11: end function
```

Secondly, we prove (ii) by proposing a reduction of the deadlock-freedom-checking problem to the local-deadlock-freedom-checking one.

Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be the supercombinator machine we try to show to be *deadlock free*, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $L = (S, \Sigma, \Delta, \hat{s})$ its induced LTS. We can check deadlock freedom for $\mathcal{S}$ by checking local-deadlock freedom for supercombinator machine $\mathcal{T}'(\mathcal{S})$, which we describe next.

$$\mathcal{T}'(\mathcal{S}) = (\langle L'_1, \ldots, L'_n \rangle, \mathcal{R}')$$

We construct $\mathcal{T}'(\mathcal{S})$ in a way that for any subsystem $ss \subset \{1 \ldots n\}$, $blocked_{ss}(s) = false$. We enforce this by always making sure some projected rule can be triggered when $ss \subset \{1 \ldots n\}$. On the other hand, when $ss = \{1 \ldots n\}$, $blocked_{\{1\ldots n\}}$ (which is equivalent to $blocked$) for $\mathcal{T}'(\mathcal{S})$ coincides with $blocked$ for $\mathcal{S}$. Therefore, checking local-deadlock freedom for $\mathcal{T}'(\mathcal{S})$ yields the same result as checking deadlock freedom for $\mathcal{S}$. Machine $\mathcal{T}'(\mathcal{S})$ modifies each component so that component i can always offer the fresh event c_i. Note, in particular, that $c_i \neq c_j$ if $i \neq j$.

$$L'_i = (S_i, \Sigma_i \cup \{c_i\}, \Delta_i \cup \{(s, c_i, s) \mid s \in S_i\}, \hat{s}_i)$$

Then, $\mathcal{T}'(\mathcal{S})$ creates new system rules for performing events c_i. Each component i is only allowed to engage on c_i (producing a system-level event c_i) if some other component also offers c_i.

$$\mathcal{R}' = \mathcal{R} \cup \bigcup_{i \in \{1\ldots n\}} \{(\{(i, c_i), (j, c_i)\}, c_i) \mid j \in \{1 \ldots n\} \wedge j \neq i\}$$

For $\mathcal{T}'(\mathcal{S})$, when considering a subsystem ss where $j \notin ss$ for some $j \in \{1 \ldots n\}$, all components $i \in ss$ can perform c_i (triggering a system-level c_i) according to the projected rules $\mathcal{R}'_{ss}$. Note that for each component i, there is a rule $r = (\{(i, c_i), (j, c_i)\}, c_i) \in \mathcal{R}'$ that requires i and j to offer c_i and yield system-level event c_i as per the definition of $\mathcal{R}'$. If $j \notin ss$ and $i \in ss$, r gives rise to a projected rule $r' = (\{(i, c_i)\}, c_i) \in \mathcal{R}'_{ss}$ that only requires the participation of i. As component i can offer event c_i in all of its states, it can perform c_i and trigger r' in any of its states, making the system progress and, hence $blocked_{ss}$ such that $ss \neq \{1 \ldots n\}$ maps any system state to *false*. Thus, no subsystems ss of $\mathcal{T}'(\mathcal{S})$ where $ss \neq \{1 \ldots n\}$ admits a local deadlock.

On the other hand, if $ss = \{1 \ldots n\}$, we have that $blocked_{\{1\ldots n\}}$ for $\mathcal{T}'(\mathcal{S})$ coincides with $blocked$ for $\mathcal{S}$. We have that $\mathcal{R}' = \mathcal{R}'_{ss}$ for $ss = \{1 \ldots n\}$. Moreover, note that the extra rules that $\mathcal{R}'$ have with respect to $\mathcal{R}$, i.e., rules $(\{(i, c_i), (j, c_i)\}, c_i)$ where i and j are components of the system such that $j \neq i$, require component i to synchronise with another component j in event c_i. However, component j can never engage in c_i (it can engage, instead, in event c_j). Hence, none of these rules can be triggered when $ss = \{1 \ldots n\}$. Since these extra rules in $\mathcal{R}'$ cannot be triggered by $\mathcal{T}'(\mathcal{S})$ when $ss = \{1 \ldots n\}$, $blocked_{\{1\ldots n\}}$ can be calculated based on the applications of rules in $\mathcal{R}$ alone. Thus, $blocked$ for $\mathcal{S}$ (which considers the application of rules in $\mathcal{R}$) coincides with $blocked_{\{1\ldots n\}}$ for $\mathcal{T}'(\mathcal{S})$ (which considers the application of rules $\mathcal{R}'$ but we know that rules in $\mathcal{R}' \setminus \mathcal{R}$ cannot be applied).

Systems $\mathcal{S}$ and $\mathcal{T}'(\mathcal{S})$ can reach the same states, as the alterations performed in $\mathcal{S}$ to create $\mathcal{T}'(\mathcal{S})$ do not affect reachability. The extra system rules requiring the participation of components in c_i events would trigger the self-loop c_i-transitions we add to components, hence creating system (loop) transitions that cause the system to remain in the same state. So, our modified system $\mathcal{T}'(\mathcal{S})$ can only explore new system states by applying the original rules $\mathcal{R}$ of $\mathcal{S}$; the extra rules $\mathcal{R}' \setminus \mathcal{R}$ make $\mathcal{T}'(\mathcal{S})$ stay in the same state.

Finally, the creation of L'_i only involves the addition of $|S_i|$-many edges, whereas $\mathcal{R}'$ involves the addition of $\mathcal{O}(n^2)$-many rules, for each c_i event it creates $n-1$-many new rules. Thus, machine $\mathcal{T}'(\mathcal{S})$ can be constructed in time polynomial on the size of $\mathcal{S}$. QED □

4. Approximate reachability using local analysis

One way to cope with the complexity of verification problems is by proposing approximate verification frameworks. As many verification tasks rely on proving that some defined set of bad states is not reachable (that is, they are based around reachability analysis), we focus on studying and proposing reachability over-approximations; approximate frameworks naturally arise from replacing exact reachability by our approximations in the verification task at hand. In this section, we study how *local analysis* and the invariants they capture can be used to create such approximations.

Often, a system property can be proved by local invariants deduced from the way small subsystems behave. In these cases, verification frameworks could benefit from examining only these subsystem instead of carrying out the costly analysis of the entire system's behaviour. This sort of analysis of some small subsystems to ensure a given property is commonly called local analysis. The first kind of reachability approximation that we propose in this work is based on the behaviour of a system's *subsystem*. A subsystem is given by a non-empty set of indices that denotes participating components. The notion of *subsystem reachability* is intended to be a means to capture and implement local analysis. We analyse the behaviour of a subsystem based on the following *subsystem projection*.

Definition 9 Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine, $\langle L_1, \ldots, L_n\rangle_{ss}$ the sequence of components resulting from removing the elements for which indices are not in ss, and e_{ss} the event tuple resulting from removing e_i if $i \notin ss$. The *subsystem projection* of machine $\mathcal{S}$ over subsystem $ss \subseteq \{1 \ldots n\} \mid ss \neq \emptyset$ is given by the following supercombinator machine:

$$\mathcal{S}_{ss} = (\langle L_1, \ldots, L_n\rangle_{ss}, \{(e_{ss}, a) \mid (e, a) \in \mathcal{R} \wedge \exists\, i \in ss \bullet e_i \neq -\})$$

This machine allows components in this subsystem to run independently from the rest of the system. Note that the projection of system rules, caused by e_{ss}, allows a subsystem's components to ignore any need for synchronisation with components that are not part of this subsystem.

We use this projection to define *subsystem reachability*.

Definition 10 Let $\mathcal{S}_{ss}$ be a supercombinator machine, resulting from the projection of $\mathcal{S}$ on subsystem ss, and $(S_{ss}, \Sigma_{ss}, \Delta_{ss}, \hat{s}_{ss})$ its induced LTS. We define $reachable_{ss}(s)$, for $s \in S_{ss}$, as the reachability predicate considering this projection's induced LTS.

Subsystem reachability can be used to over-approximate a system's reachability. If a subsystem projection cannot reach a state, it must be the case that any system state that extends this projection state is unreachable. A system state extends a projection state, involving subsystem ss, if they share the same component states for the components in ss. We use predicate $reach_{ss}(s)$, defined next, to capture this approximation.

Definition 11 Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and ss a given subsystem of $\mathcal{S}$. For $s \in S$, we define $reach_{ss}(s)$ as $reachable_{ss}(s_{ss})$. It lifts the predicate $reachable_{ss}$ which applies to states of subsystem ss to the states of the system $\mathcal{S}$ itself. For $s = (s_1, \ldots, s_n)$, s_{ss} creates a state-tuple with the $|ss|$ elements s_i where $i \in ss$.

The over-approximating nature of this predicate is a consequence of our projection's rules discarding the participation of components outside the subsystem. This discarding can be seen as placing the subsystem in an ideal synchronisation context, where the original components outside the subsystem are replaced by components that always offer all events. So, if a projection cannot reach a state with all this help, no combination of components (and, in particular, the original components outside the subsystem) can help the subsystem in reaching this state. By contraposition, it must be the case that if a state is reached by the system, the projected state must be reachable in the context of the projection.

Lemma 6 $reachable(s) \Rightarrow reach_{ss}(s)$

Proof This can be proved by induction on the size of the path $\hat{s} \xrightarrow{tr} s$. We use ss to denote the projection's subsystem and $\rightarrow_{ss}$ to denote the path predicate for the LTS induced by this projection. The base case is trivial as $\hat{s} \xrightarrow{\langle\rangle} \hat{s}$ and $\hat{s}_{ss} \xrightarrow{\langle\rangle}_{ss} \hat{s}_{ss}$. For the inductive case, assuming $(i.h.)$ $\hat{s} \xrightarrow{tr} s \Rightarrow \hat{s}_{ss} \xrightarrow{tr'}_{ss} s_{ss}$ and $\hat{s} \xrightarrow{tr\hat{}\langle a\rangle} s$, we split our proof into two cases:

- Some component in ss participates in performing event a. Let us assume without loss of generality that r is the rule that involves components in ss and leads to the performance of a. According to our projection definition, there must be a projected rule r' that requires the same event for these components in ss and also performs a. So, by $(i.h.)$, we can deduce that $\hat{s}_{ss} \xrightarrow{tr'\hat{}\langle a\rangle}_{ss} s_{ss}$.
- No component in ss participates in performing event a. Let us assume without loss of generality that $\hat{s} \xrightarrow{tr} s' \xrightarrow{a} s$. Since no component in ss participates on performing a, $s'_{ss} = s_{ss}$. So, by $(i.h.)$ $\hat{s}_{ss} \xrightarrow{tr'}_{ss} s_{ss}$.

QED □

We use subsystem reachability to systematically create a family of reachability over-approximations. For integer $k \geq 1$, we propose the notion of *k-reachability* that combines subsystem-reachability approximations for some (up-to-)k-sized subsystems. The approximations in this family should be reasonably precise and could be readily used to construct a fully automatic verification framework. Instead of placing the burden of choosing the subsystems that are pertinent in proving a given property on the user, this family should offer some guidance in (and a ready-to-use strategy for) picking some generally relevant sets of subsystems. We choose the subsystems that take part in our k-reachability approximation based on how components are connected. To analyse these connections, we rely on the system's *communication graph*.

Definition 12 Let $\mathcal{S}$ be a supercombinator machine with n components and rules $\mathcal{R}$. $\mathcal{S}$'s communication graph CG is an undirected graph where nodes are component indices and there is an edge between two component indices if they participate together on a rule: $CG = (\{1 \ldots n\}, \{(i, j) \mid i, j \in \{1 \ldots n\} \wedge ((e_1, \ldots, e_n), a) \in \mathcal{R} \wedge e_i \neq - \wedge e_j \neq -\})$.

We use connections to identify which components interact (and, consequently, directly interfere) with each other. So, we pick sets of k closely-interacting components, namely, we choose all k-sized subsystems for which component indices induce a *connected* communication subgraph. It does not make sense to analyse a subsystem that induces a disconnected communication graph because the same reachability guarantees can be obtained by analysing the connected subcomponents of this graph separately. Since these two subcomponents do not share rules, they engage on transitions completely independently. Also, examining these subcomponents independently leads to a more efficient reachability analysis. Moreover, note that if some (graph-theoretic) connected component[1] of the communication graph involves only $nc < k$ system components, the system components in this subgraph can never be part of a set of k closely-interacting system components. Hence, in such cases, we add the nc-sized set of system components in this subgraph to our analysis. We collect subsystems in the set SS_k.

Definition 13 Let $\mathcal{S}$ be a supercombinator machine, CG its communication graph, $\{CG_1, \ldots, CG_m\}$ this graph's connected components where $CG_i = (V_i, E_i)$, and $k_i = min(|V_i|, k)$. SS_k selects closely-interacting subsystems of size (up-to-)k:

$$SS_k = \bigcup_{i \in \{1 \ldots m\}} SS(i)$$

- $SS(i) = \{ss \mid ss \subseteq V_i \wedge |ss| = k_i \wedge CG_{ss} \textit{ is connected}\}$
- CG_{ss} gives the subgraph of CG involving vertices in ss

We conjoin the reachability approximations for all these subsystems in an effort to create a tight approximation for the entire system. For a given k, this combination gives rise to the approximation (and predicate) $reach_k$, which captures our notion of k-reachability.

[1] A connected component of an undirected graph is a subgraph for which all nodes are mutually reachable through a path and disconnected from any other nodes of the underlying undirected graph.

Definition 14 Let $\mathcal{S}$ be a supercombinator machine, and SS_k its set of subsystems involving closely-interacting components.

$$reach_k(s) = \bigwedge_{ss \in SS_k} reach_{ss}(s)$$

The fact that this predicate soundly approximates reachability for the systems under analysis follows from Lemma 6 and the fact that a conjunction of over-approximations is also an over-approximation.

Lemma 7 $reachable(s) \Rightarrow reach_k(s)$

We can test whether $reach_k(s)$ holds for a given state s and integer k in polynomial time on the size of the supercombinator machine being analysed.

Lemma 8 Let $\mathcal{S}$ be a supercombinator machine with n components and rules $\mathcal{R}$, and L_{Max} be the component with the largest LTS. For a given state s of this system and an integer k, we can test $reach_k(s)$ in time $\mathcal{O}(n^k \cdot |L_{Max}|^{k+1} \cdot |\mathcal{R}|)$.

Proof There are at most $\binom{n}{k} \in \mathcal{O}(n^k)$ subsystems in SS_k; this maximum is reached for systems that have a fully-connected communication graph. Depth-first search can be used to find the subsystems in SS_k.

For a $ss \in SS_k$, we can test $reach_{ss}(s)$ in time $\mathcal{O}((\prod_{i \in ss} |L_i|) \cdot |\mathcal{R}| \cdot (\sum_{i \in ss} |L_i|))$ by explicitly constructing and exploring the state space of this subsystem's projection. We can do that by enumerating its states and using rule application to create its transitions. A rule application takes $\mathcal{O}(\sum_{i \in ss} |L_i|)$ time; to check whether it can be applied to a given state, we might need to check each transition of each component. So, checking all rule applications for a given state should take $\mathcal{O}(|\mathcal{R}| \cdot (\sum_{i \in ss} |L_i|))$ time. Enumerating all states of this projection takes $\mathcal{O}(\prod_{i \in ss} |L_i|)$. To create all transitions, we might need to carry out these rule applications for all states and that takes $\mathcal{O}((\prod_{i \in ss} |L_i|) \cdot |\mathcal{R}| \cdot (\sum_{i \in ss} |L_i|))$ time.

Therefore, it takes $\mathcal{O}(\sum_{ss \in SS_k}((\prod_{i \in ss} |L_i|) \cdot |\mathcal{R}| \cdot (\sum_{i \in ss} |L_i|)))$ time to test $reach_{ss}(s)$ for all $ss \in SS_k$. As $|SS_k|$ is bounded by n^k, $\prod_{i \in ss} |L_i|$ by $|L_{Max}|^k$, and $\sum_{i \in ss} |L_i|$ by $k \cdot |L_{Max}|$, testing $reach_k(s)$ takes $\mathcal{O}(n^k \cdot |L_{Max}|^{k+1} \cdot |\mathcal{R}|)$ time. QED □

It is also the case that the higher the k is, the more precise is the reachability approximation $reach_k$. Intuitively, by taking larger subsystems, this approximation can support a better understanding of the overall behaviour of the system at the expense of more calculations.

Lemma 9 $reach_{k+1}(s) \Rightarrow reach_k(s)$

Proof We prove this by contradiction. Let us assume that $s = (s_1, \ldots, s_n)$ is a system state such that $reach_{k+1}(s)$ and $\neg reach_k(s)$. Since $\neg reach_k(s)$, let us say $ss \in SS_k$ is a subsystem such that (i) $\neg reach_{ss}(s)$. There are two cases to consider: either $ss \in SS_{k+1}$ (if components in ss are disconnected from the other components of the system), or there is some ss' such that $ss \subseteq ss'$ and $ss' \in SS_{k+1}$ (if components in ss are connected to another component). In case $ss \in SS_{k+1}$, (i) trivially implies $\neg reach_{k+1}(s)$, contradicting our assumption. In the other case, let us say that ss' is a subsystem such that (ii) $ss \subseteq ss'$ and (iii) $ss' \in SS_{k+1}$. Our projection definition ensures that if $ss \subseteq ss'$, $\neg reach_{ss}(s)$ implies $\neg reach_{ss'}(s)$. So, thanks to (i) and (ii), $\neg reach_{ss'}(s)$ holds, and (iii) implies that $\neg reach_{k+1}(s)$, a contradiction. QED □

Any approximation in this family, other than k equalling the size of the system (or more precisely the size of the largest component of the communication graph), is intrinsically incomplete/imprecise thanks to the pure use of local analysis. For any given k, our notion of k-reachability is unable to show that a system respects a given property if it emerges from the behaviour of a subsystem of size greater than k. This limitation is inherent to any verification framework purely on local analysis; this limitation is acceptable as long as k-reachability gives rise to an efficient verification framework. Since it is difficult to generally anticipate what is the lowest k such that k-reachability can prove the property at hand, one could, in practice, devise a heuristic by which a few increasingly large ks are chosen and the corresponding reachability predicate constructed. If none of them give rise to a reachability predicate that is strong enough to prove the property at hand, a precise approach is used instead.

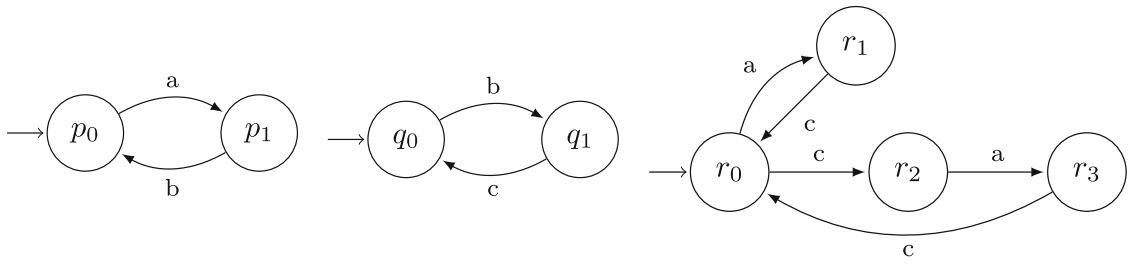

Fig. 3. LTSs of components L_1, L_2 and L_3, respectively

5. *Pair*: 2-reachability for deadlock and local-deadlock freedom

In this section, we demonstrate how local analysis can be used to create an effective verification framework. We use 2-reachability to verify deadlock and local-deadlock freedom for concurrent systems. We chose $k = 2$ as 1-reachability is too coarse and should not be able to prove interesting properties of concurrent systems; 1-reachability does not account for interactions between components so it can only prove properties that emerge from the individual behaviour of components. We call our framework *Pair*, as it is based on the analysis of pairs of components to approximate reachability. Instead of looking for cycles of dependencies between components of a system as traditional approximate frameworks do, *Pair* looks for states of the system that are 2-reachable and in which all further actions are blocked (or, locally blocked); a system state of this sort is considered a potential deadlock (or, local deadlock) and we call it a *Pair candidate* (or, *Pair local candidate*).

Definition 15 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 $pair\_candidate(s) = reach_2(s) \wedge blocked(s)$ holds;
- a *Pair local candidate* if and only if $pair\_local\_candidate(s) = reach_2(s) \wedge locally\_blocked(s)$ holds.

As a *Pair* candidate is also a *Pair* local candidate, the following holds.

Lemma 10 If a system is free of *Pair* local candidates, it must also be free of *Pair* candidates.

Our framework is sound, as absence of *Pair* candidates implies deadlock freedom; the same holds for local candidates and local-deadlock freedom. This follows from the fact that $reach_2(s)$ approximates reachability as per Lemma 7.

Theorem 3 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. If $\mathcal{S}$ is *Pair*-candidate free, it must be also deadlock free. Similarly, if $\mathcal{S}$ is *Pair*-local-candidate free, it must be also local-deadlock free.

These criteria will be shown to be more accurate than many of the conditions checked by current approximate frameworks, but it remains incomplete as it relies on local analysis to approximate reachability; there may well be some *Pair* (local-)candidate that is not actually reachable.

Example 2 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 Fig. 3 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 2-reachable and blocked, and consequently locally blocked, but not reachable. Thus, it constitutes both a *Pair* candidate and local candidate but not a true deadlock or a local deadlock.

Our framework looks for a blocked (locally blocked) state amongst the system states that are 2-reachable instead of going through the system's exact state space. Since we exactly check whether a state is blocked (locally-blocked), our method is imprecise only as far as reachability is concerned. So, false negatives can only arise from the fact that 2-reachability was unable to prove that a candidate (local-candidate) is unreachable.

5.1. Precision of *Pair*

In this section, we analyse the precision of our approach by comparing it to the traditional approximate approaches that are based on the detection of cycles of dependencies between components. We compare *Pair* against the *SDD framework* developed by Martin in [Mar96]. We chose Martin's SDD framework for four reasons. Firstly, it inspired our study on over-approximations for deadlock-freedom checking and the creation of *Pair*. Secondly, it is a typical example of a framework based on proving absence of ungranted-requests (i.e. dependencies) cycles. Thirdly, its underlying formalism is very close to ours. Finally, it can show absence of such cycles (and consequently local-deadlock and deadlock freedom) for some relevant classes of systems. Martin has shown, for instance, that his framework can prove local-deadlock freedom for some systems implementing two very well-known interaction paradigms: the *resource-allocation* and *client-server* paradigms.

In that work, the local properties are derived from the analysis of pairs of components through the following supercombinator machine.

Definition 16 Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. The *pairwise machine* $\mathcal{S}_{i,j}$ is used to analyse the interactions of components i and j.

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

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 $\mathcal{S}_{i,j}$, and $ungranted\_request_{i,j}(s, s')$ holds when, in their respective states (i in s and j in s'), component i is willing to interact (i.e. engage on a rule) with j (according to $\mathcal{S}_{i,j}$) but j is unable to do so.

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

Definition 17 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 SDD candidate is a sequence of component states $c \in \mathcal{U}^*$ where for all $i \in \{0 \ldots |c| - 1\}$, 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 $\oplus$ is addition modulo $|c|$.

This method can show (local-)deadlock freedom very efficiently: we can decide whether or not a digraph has cycles in linear time using a modified depth-first search. This efficiency, however, comes with a price as the use of such 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 necessarily having a single node per component—after all, no component can be in two different states in a single (local-)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 (cycle-wise) component states not tested for that might eliminate some SDD candidate. For instance, a cycle where two non-adjacent component states are not mutually reachable (or, alternatively can actually synchronise with each other) cannot represent a true (local-)deadlock and so it should not be considered a SDD candidate. Finally, a cycle, as a necessary condition, is bound to arise in some (local-)deadlock-free systems. Thus, in such cases, this framework is ineffective. The reason why these sources of imprecision are not addressed is that these methods look for polynomially checkable conditions for guaranteeing (local-)deadlock freedom and tackling any of these sources of imprecision is likely to make the problem of finding a candidate *NP*-hard.

The combination of 2-reachability and exactly checking for blocked system states makes *Pair*'s analysis of reachability and the blocking conditions more precise than SDD's. *Pair* uses an *exact* characterisation for blocked states instead of the imprecise cycle-of-dependencies one. As for reachability, SDD only requires pairs of component states adjacent in the cycle to be mutually reachable. *Pair*, however, enforces this for all pairs of components in a blocked system state. The improvement in precision means that none of the potential sources of imprecision highlighted in the previous paragraph affects *Pair*. In fact, only its reachability analysis

is imprecise. The use of 2-reachability means that it cannot prove (local-)deadlock freedom if this property depends on some reachability invariant of triples or larger combinations of components.

This informal comparison can be formalised to show that in fact *Pair* is strictly more precise than SDD. For this comparison, as required by SDD, we assume that supercombinator machines' components are deadlock free. Under this assumption, in a blocked state, all components must be willing to interact with another component that, in turn, does not want to interact back. So, each component state in this blocked system state must be the origin of some ungranted request leading to another component state in this system state. As we only have finitely many component states in this system state, there must be a cycle amongst them.

Lemma 11 (Strengthening of Theorem 1 in [Mar96]) If $blocked_{ss}(s)$ holds for some system state $s = (s_1, \ldots, s_n)$ and subsystem ss, there must be a cycle of ungranted requests amongst (induced by) component states s_i for $i \in ss$.

It follows from this lemma and the fact that *Pair* local candidates are 2-reachable that a *Pair* local candidate must induce a SDD candidate. Being locally blocked, this system state must induce a cycle of ungranted requests for which pairs of components are pairwise reachable as per 2-reachability.

Lemma 12 If a system state s is a *Pair* local candidate, its component states must induce a SDD candidate.

As a *Pair* candidate is also a *Pair* local candidate, it follows that it also induces a SDD candidate.

Corollary 1 A *Pair* candidate induces a SDD candidate.

Moreover, this lemma also implies by contraposition that a SDD-candidate-free system must also be *Pair*-local-candidate free and, consequently, *Pair*-candidate free.

Corollary 2 If a system is SDD-candidate free, it must also be free of *Pair* candidates and local-candidates.

These results prove that our framework shows local-deadlock and deadlock freedom for any system SDD does. Hence, we can reuse any result about the precision (i.e. relative completeness) of SDD for *Pair*. For example, Martin has proposed a number of design rules, based on [RD87], that can be used to construct local-deadlock-free systems which can be proved so by SDD. We briefly and informally introduce two classes of systems, namely, resource-allocation and client-server systems, that can be constructed using these rules. For more details on these rules see [Mar96].

The resource-allocation rule can be used to create systems where some user components need to acquire some shared resources in order to carry out a task. This rule ensures that users never get blocked due to some cyclic wait: a user is waiting for another user to release a resource that in turn is waiting for another user leading all the way back to the initial waiting user. It ensures users respect a linear order in acquiring resources so no cyclic wait, and consequently no local-deadlock, can arise. This rule was initially proposed by the operating-system community to prevent deadlocks due to the ill allocation of operating system's resources to system processes [CES71].

A system in this class is composed of *user* and *resource* components. Resources can be acquired and released by users and users can acquire resources respecting a linear order on resources. A user only tries to acquire resources that are higher in this order than the ones it already holds. In such a system, cycles of dependencies are broken, and consequently a local-deadlock is prevented, by the acquisition of resources respecting this linear order. This rule has been similarly formalised in other works [SD82, RD87].

The client-server rule can be used to create systems where components interact in a request-response fashion. This rule ensures that components never get blocked due to some cyclic wait of the form: a component is waiting for some server component that in turn is waiting for another server component leading all the way back to the initial waiting component. To prevent this sort of cyclic wait, it ensures components only make requests to other components respecting an underlying request-response structure that must be cycle free.

Each component in such a system can alternate in behaving as a client or a server performing request and response actions. As a client the component must be able to request some of its server after which it must be ready to receive any response back, and as a server it must be waiting for a request from any of its clients after which it can issue some response. Also, the *client-to* digraph must be cycle free; this digraph has an arrow from i to j if component i can behave as a client (i.e. make a request) to j. In such a system, an ungranted request coincides with edges of the *client-to* digraph and hence cannot be part of a cycle.

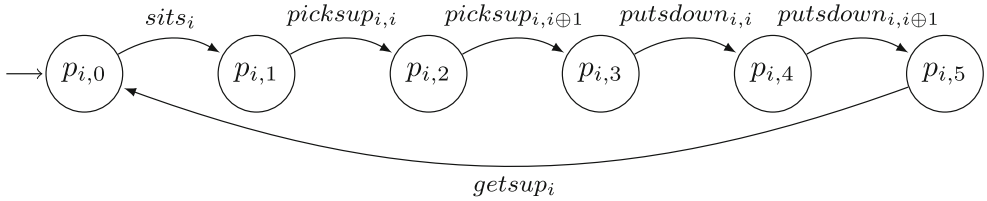

Fig. 4. LTS of philosopher i

The results presented so far show that *Pair* is at least as accurate as SDD; we extend them to show that *Pair* is strictly more precise than SDD. This strict improvement can be informally deduced from the way these frameworks operate. While SDD looks through cycles of ungranted requests to show they do not constitute a real (local-)deadlock by finding a pair of *adjacent* component states that is mutually unreachable, *Pair* looks through (locally-)blocked system states and show they are not true (local-)deadlocks by finding *any* pair of component states that is mutually unreachable. So, for instance, a system that possesses a cycle of ungranted requests where all adjacent pairs of component states are mutually reachable but a pair of non-adjacent component states can interact or are mutually unreachable is proved (local-)deadlock free by *Pair* but not by SDD.

This analysis points to a class of relevant systems that can be proved local-deadlock free by *Pair* but not by SDD: the class of *non-hereditary deadlock-free systems*. These systems have a subsystem that can deadlock and a guard-like component that leads the subsystem away from this blocked state. SDD cannot show such systems local-deadlock free since 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 as per Lemma 12. While SDD only allows cycles of ungranted requests to be broken (i.e. prevented from occurring) by component states within the cycle, *Pair* can be understood as allowing also component states outside the cycle to break it, hence its ability to tackle such systems. Roughly speaking, SDD can be seen as a method designed to verify *hereditary* deadlock-free systems, whereas our method can prove (local-)deadlock freedom for both hereditary and non-hereditary deadlock-free systems, such as in the following example.

While local-deadlock freedom examines whether subsystems are blocked in the context of the system, hereditary deadlock freedom asks whether subsystems are deadlock free by themselves. So, a subsystem state where it deadlocks is a violation for hereditary deadlock freedom, regardless of whether a component external to this subsystem can prevent this deadlock. Local-deadlock freedom, on the other hand, accounts for possible external components that can prevent such a deadlock. Note that hereditary deadlock freedom implies local deadlock freedom. Furthermore, hereditary deadlock freedom is not really useful other than as a piece of system analysis, whereas local deadlock freedom is an important practical property. Finally, we point out that many concurrent systems are not hereditary deadlock free; systems that have components implementing mutual exclusion algorithms or semaphores to prevent some subsystem from reaching undesired states are fairly commonplace in the concurrency literature.

Example 3 This well-known system is composed of three different types of 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 Fig. 4 (5). We use $\oplus$ to denote 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 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 $S \in \mathbb{P}(\{1 \dots N\}) \backslash \{\{1 \dots N\}\}$. Its transitions are created as depicted in Fig. 5, 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 (local-)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:

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

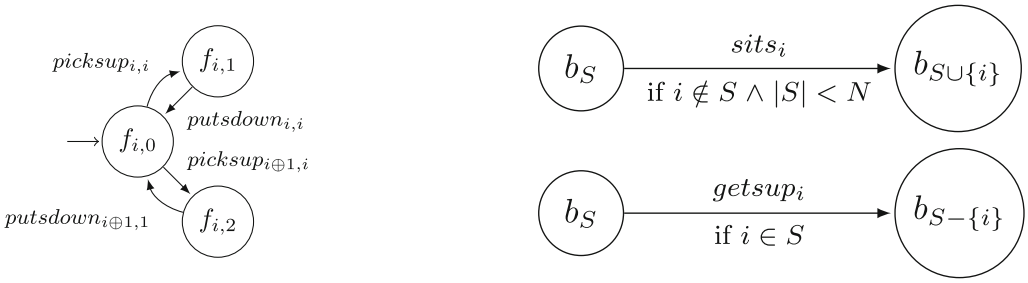

Fig. 5. LTS of fork i and transitions of the butler process

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.

Even though the *Pair* method is better than traditional approximate methods for checking deadlock freedom, it is, nonetheless, still an approximate framework in itself. *Pair* is unable to show that a system is (local-)deadlock free if (local-)deadlock freedom depends on some reachability invariant of triples or larger combinations of components. That is, if a blocked system state is found and it can only be shown unreachable due to the combined behaviour of some subsystem involving more than a pair of components, then *Pair* will fail to rule this state out as (local-)deadlock candidate and to show, consequently, that the system is (local-)deadlock free.

5.2. Complexity of *Pair*

The improvement in precision that *Pair* makes over traditional approximate techniques comes with a price. While detecting *Pair* candidates and *Pair* local candidates are NP-hard problems, traditional approaches detect candidates in polynomial time. Addressing any of the sources of imprecision that *Pair* tackles with respect to SDD, for instance, is likely to turn candidate detection into a NP-hard problem. This argument seems to be the reason behind traditional frameworks not attempting to address any of them. Moreover, unsurprisingly, the use of reachability approximations instead of exact reachability makes this problem more tractable than exact (local-)deadlock checking.

Next, we show that the problem of detecting a *Pair* candidate is *NP*-complete, whereas its counterpart for *Pair* local candidates is *NP*-hard, and we discuss some implications of these results.

Theorem 4 Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. The problem of deciding $\exists\, s \in S \bullet pair\_candidate(s)$ is *NP-complete*.

Proof We show that this problem is (i) in *NP* and (ii) *NP*-hard.

We show (i) by establishing that, for a given system state s, $pair\_candidate(s)$ can be verified in $\mathcal{O}(n^2 \cdot |L_{Max}|^3 \cdot |\mathcal{R}|)$ time, where n and $\mathcal{R}$ give the number of components and the rules of the system, respectively, and $|L_{Max}|$ the size of the largest component LTS. $reach_2$ can be checked in $\mathcal{O}(n^2 \cdot |L_{Max}|^3 \cdot |R|)$ time, as per Lemma 8, while $blocked(s)$ can be checked in $\mathcal{O}(|\mathcal{R}| \cdot n \cdot |L_{Max}|)$ time. To check $blocked(s)$, one can simply check whether a rule can be applied to s.

To demonstrate (ii), we present a polynomial-time 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 18 Given a CNF (i.e. Conjunctive Normal Form) boolean formula $\mathcal{F}$ with boolean variables $x_1, \ldots, x_m$, the *CNF-SAT* problem consists of finding an assignment to the m boolean variables so that $\mathcal{F}$ holds, i.e.: checking $\exists\, x_1, \ldots, x_m \in \{true, false\} \bullet \mathcal{F}$.

Let $\mathcal{F}$ be a CNF boolean formula with m boolean variables $x_1, \ldots, x_m$ and n clauses $\mathcal{F}_1, \ldots, \mathcal{F}_n$, where clause $\mathcal{F}_i$ has n_i literals $\mathcal{F}_{i,1}, \ldots, \mathcal{F}_{i,n_i}$. We assume without loss of generality that this formula has at least one clause and that all variables are present in some clause of the formula. Our reduction translates this formula into the following *triple-disjoint* supercombinator machine such that it has a *Pair*-candidate if and only if the formula is satisfiable.

$$\mathcal{S} = (\langle F_1, \ldots, F_n, X_1, \ldots, X_m \rangle, \mathcal{R})$$

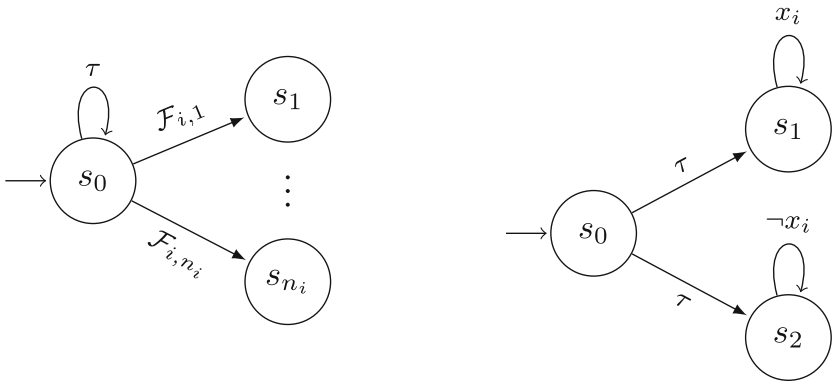

Fig. 6. LTSs F_i and X_i respectively

In this machine, component F_i captures the satisfiability of clause $\mathcal{F}_i$, whereas component X_i models the assignment of boolean variable x_i. We use literals x_j and $\neg x_j$ as events that denote whether variable x_i has been assigned to *true* and *false*, respectively. In particular, $\mathcal{F}_{i,j}$ are events. Satisfiability of $\mathcal{F}_i$ is captured by creating a transition, in F_i, with event x_j ($\neg x_j$) to a *terminal* deadlocked state for each literal x_j ($\neg x_j$) in $\mathcal{F}_i$. So, a terminal state is only reached if the clause has been satisfied (i.e. a literal has been satisfied). Next, we present the definitions of component F_i and X_i and their graphical representation in Fig. 6.

$F_i = (S, \Sigma, \Delta, \hat{s})$, where:

- $S = \{s_0, \ldots, s_{n_i}\}$
- $\Sigma = \{\mathcal{F}_{i,j} \mid j \in \{1 \ldots n_i\}\} \cup \{\tau\}$
- $\Delta = \{(s_0, \tau, s_0)\} \cup \{(s_0, \mathcal{F}_{i,j}, s_j) \mid j \in \{1 \ldots n_i\}\}$
- $\hat{s} = s_0$

$X_i = (S, \Sigma, \Delta, \hat{s})$, where:

- $S = \{s_0, s_1, s_2\}$
- $\Sigma = \{\tau, x_i, \neg x_i\}$
- $\Delta = \{(s_0, \tau, s_1), (s_0, \tau, s_2), (s_1, x_i, s_1), (s_2, \neg x_i, s_2)\}$
- $\hat{s} = s_0$

The set of rules provided enables the synchronisation between variable components and clauses components so that the satisfiability of the clauses is guided by the assignment of variables. We represent an event tuple as a set of pairs. Also, we use $F_i = i$ and $X_{i,j} = n + m_{i,j}$ to denote the positions of components in the event tuple, where $m_{i,j}$ denotes the index of the boolean variable in literal $\mathcal{F}_{i,j}$, so if $\mathcal{F}_{i,j} = x_k$ or $\mathcal{F}_{i,j} = \neg x_k$ then $m_{i,j} = k$. Therefore, for instance, the rule $(\{(i, \tau)\}, \tau)$ triggers the system event τ if component i performs τ, whereas the rule $(\{(F_i, \mathcal{F}_{i,j}), (X_{i,j}, \mathcal{F}_{i,j})\}, \mathcal{F}_{i,j})$ requires components F_i and $X_{i,j}$ to synchronise on event $\mathcal{F}_{i,j}$ to produce the system event $\mathcal{F}_{i,j}$.

$$\mathcal{R} = \{(\{(i, \tau)\}, \tau) \mid i \in \{1 \ldots n + m\}\} \cup \bigcup_{i \in \{1 \ldots n\}} \{(\{(F_i, \mathcal{F}_{i,j}), (X_{i,j}, \mathcal{F}_{i,j})\}, \mathcal{F}_{i,j}) \mid j \in \{1 \ldots n_i\}\}$$

From $\mathcal{S}$'s definition, we can see this supercombinator machine can be constructed in polynomial time on the size of the formula $\mathcal{F}$. All components can be constructed in time proportional to $|\mathcal{F}|$. Each component X_i can be constructed in time $\mathcal{O}(1)$, whereas components F_i can be constructed in time $\mathcal{O}(|\mathcal{F}_i|)$. Rules $\mathcal{R}$ can be constructed in time $\mathcal{O}(|\mathcal{F}|)$, as it creates a rule per literal and a τ-rule per component.

This machine also precisely captures satisfiability for the associated boolean formula, namely, there is a *Pair* candidate for this supercombinator machine *iff* the corresponding boolean formula is satisfiable.

If the formula is satisfiable, $\mathcal{S}$ has a *Pair*-candidate. Let $\mathcal{A}$ be a satisfying assignment for the formula. System $\mathcal{S}$ can reach a state where components X_i are in states respecting their valuation $\mathcal{A}(x_i)$, and components F_i are in terminal states. This state is reachable as components X_i can simply τ-transition to their respective

valuation states, and because $\mathcal{A}$ is a satisfying assignment, each component F_i must be able to synchronise with some X_i to reach a terminal state. If a state is reachable it is also 2-reachable. This state is also blocked since all F_i components are in terminal states and all components X_i are in states that can only trigger system rules involving the participation of at least one F_i component.

If $\mathcal{S}$ has a *Pair* candidate, then the formula is satisfiable. Let s be the state representing a *Pair* candidate. Based on the definition of $\mathcal{S}$ and since it is blocked, it must have all components X_i in some valuation state and all components F_i in a terminal state. The 2-reachability enforces that whichever transition F_i took to reach its terminal state, it must have been in agreement with the corresponding valuation state of X_i in s. Therefore, since all components F_i are in terminal states thanks to the valuation states of components X_i, the valuation states of components X_i provide a satisfying assignment to $\mathcal{F}$. QED □

The hardness part of this proof can be generalised, leading to some interesting results. For the system that we propose in that reduction, $reach_2(s)$ precisely captures reachability. Therefore, any approximation more precise than $reach_2(s)$ also precisely determines reachability, and that same reduction can be used to show that the problem of detecting deadlock candidates for any candidate definition where we replace $reach_2(s)$ by a better approximation must be *NP*-hard.

Corollary 3 Let $reach(s)$ be a reachability over-approximation. If $reach(s) \Rightarrow reach_2(s)$ then the problem of detecting a deadlock candidate s such that $reach(s) \wedge blocked(s)$ is *NP*-hard.

This corollary, Lemma 8 and the fact that $blocked(s)$ can be decided in polynomial time, as shown in the above proof, imply that any deadlock candidate formulation using k-reachability, for $k \geq 2$, instead of exact reachability gives rise to an *NP*-complete candidate detection problem.

Corollary 4 For integer $k \geq 2$, the problem of detecting a deadlock candidate s such that $reach_k(s) \wedge blocked(s)$ is *NP*-complete.

For the problem of detecting *Pair* local candidates, we only show *NP*-hardness.

Theorem 5 Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. The problem of deciding $\exists\, s \in S \bullet pair\_local\_candidate(s)$ is *NP-hard*.

Proof To show that this problem is *NP*-hard, we reduce detecting a *Pair* candidate to it. In fact, we can achieve this reduction by detecting *Pair* local candidates using the modified machine $\mathcal{T}'(\mathcal{S})$ introduced in the proof of Theorem 2. As we discuss there, this modified machine can be created in polynomial time on the size of $\mathcal{S}$ and the modifications it performs on $\mathcal{S}$ make the notion of locally blocked for this modified machine coincide with the notion of blocked for the original one. Moreover, they do not affect reachability or 2-reachability. So, checking for *Pair* local candidates for this modified machine must yield the same result as checking for *Pair* candidate on the original one. QED □

These results build on our precision discussion. While traditional approaches are based on a polynomially checkable detection problem, *Pair* is based on a *co-NP*-hard problem: showing the absence of *Pair* candidates and local candidates. So, our frameworks should prove (local-)deadlock freedom for a different class of deadlock-free systems. Furthermore, as exact deadlock-freedom checking is *PSPACE*-complete and our *Pair* candidate detection is *only NP*-complete, it should be (and it is) the case our frameworks proves deadlock freedom for a different (and smaller) class of deadlock-free systems. Furthermore, given our general current understanding about these complexity classes, the computational tractability of our *Pair*-candidate detection problem should be in between candidate detection for traditional approaches, such as SDD, and precise deadlock detection.

5.3. *Pair*-candidate detection via SAT solving

The problem of detecting a *Pair* (local) candidate is *NP*-hard. So, currently, we only know of deterministic procedures that take exponential time to solve it. There have been, however, some remarkable advances in proposing efficient procedures to solve the propositional satisfiability (SAT) problem. So, in an attempt to efficiently tackle *Pair*-(local-)candidate detection, we propose an implementation for our framework where we translate *Pair*-(local-)candidate detection into propositional satisfiability, which can later be checked by a SAT solver.

Given a supercombinator machine as an input, our procedure creates propositional formulæ *Candidate* or *LocalCandidate*, depending on whether deadlock freedom or local-deadlock freedom, respectively, is being checked. They rely on variables $st_{i,s}$ to capture whether state s of component i is part of a *Pair* (local) candidate: the variables $st_{i,s}$ assigned to *true* in a satisfying assignment correspond to a combination of component states that is a *Pair* (local) candidate. If *Candidate* (*LocalCandidate*) is unsatisfiable, however, the input system must be *Pair*-candidate (*Pair*-local-candidate) free. In this section, we assume that $\mathcal{S} = (\mathcal{L}, \mathcal{R})$ is the input supercombinator machine we are translating, where $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ and $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$.

Formulæ *Candidate* and *LocalCandidate* are composed of three sub-formulæ each.

$$Candidate = State \wedge Reach_2 \wedge Blocked$$

$$LocalCandidate = State \wedge Reach_2 \wedge LocallyBlocked$$

The sub-formula *State* simply ensures that the variables $st_{i,s}$ assigned to *true* form a valid system state, i.e. one component state per component is assigned to *true*.

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

Sub-formula $Reach_2$ captures the approximation $reach_2$. To describe this sub-formula (and the next one), we introduce the following notation for convenience. S_{ss} gives the state space of subsystem ss. We represent subsystem states $sst \in S_{ss}$ by a set of pairs, as opposed to the traditional tuple representation, where $(i, s) \in sst$ denotes that state s of component i belongs to this subsystem state. To capture $Reach_2$, we examine the state space of subsystems in SS_2 (i.e. subsystems involving at most 2 components) and disallow unreachable combinations of component states.

$$Reach_2 = \bigwedge_{ss \in SS_2} \bigwedge_{\substack{sst \in S_{ss} \wedge \\ \neg reachable_{ss}(sst)}} \Big(\bigvee_{(i,s) \in sst} \neg st_{i,s} \Big)$$

The sub-formula *Blocked* captures the *blocked* predicate. We assume triple-disjointness of the input system in encoding this sub-formula; this is the only definition in our framework that formally depends on this assumption. Thanks to triple disjointness, we can capture whether a system state is blocked by analysing only individual and pairs of components–after all, system transitions only involve either the participation of a single component or a pair of them. To encode this blocking requirement, we use yet a different set of projected rules: $\mathcal{R} \restriction ss = \{(e, a) \mid (e, a) \in \mathcal{R} \wedge \forall\, i \in \{1 \ldots n\} \bullet (i \notin ss \wedge e_i = -) \vee (i \in ss \wedge e_i \neq -)\}$; we analyse this projection for subsystems with an individual ($ss \in SS_1$) or a pair of communicating components ($ss \in SS_2$). Unlike the other projections, this one does not truncate rules; it exactly projects (copies) rules that require the exact participation of components in ss. We use $s \xrightarrow{\mathcal{R} \restriction ss}$ to denote the existence of a transition from s in the LTS induced by $(\mathcal{L}, \mathcal{R} \restriction ss)$. Our formula forbids component states that can participate in some rule from holding simultaneously. Note that to calculate $s \xrightarrow{\mathcal{R} \restriction ss}$ we examine the state space of individual and pairs of components, as $ss \in SS_1 \cup SS_2$.

$$Blocked = \bigwedge_{ss \in SS_1 \cup SS_2} \bigwedge_{\substack{sst \in S_{ss} \wedge \\ sst \xrightarrow{\mathcal{R} \restriction ss}}} \Big(\bigvee_{(i,s) \in sst} \neg st_{i,s} \Big)$$

To encode *LocallyBlocked* sub-formulæ, we introduce new variables p_i for $i \in \{1 \ldots n\}$ to account for the participation of component i in the subsystem being analysed: assigning p_i to *true* means component i is part of the subsystem under analysis. So, the sort of existential quantification SAT does on p_i variables translates to the quantification on subsystems our *locally-blocked* requires; we add constraint $\bigvee_{i \in \{1 \ldots n\}} p_i$ to prevent quantification over the empty subsystem. For subsystem ss, it captures the $blocked_{ss}$ predicate, i.e. component states assigned to true in a satisfying assignment form a state in which subsystem ss is blocked.

$$LocallyBlocked = \bigwedge_{ss \in SS_1 \cup SS_2} \bigwedge_{\substack{sst \in S_{ss} \wedge \\ sst \xrightarrow{\mathcal{R} \restriction ss}}} on(ss) \Rightarrow \Big(\bigvee_{(i,s) \in sst} (p_i \wedge \neg st_{i,s}) \Big)$$

The constraint $on(ss) = \bigvee_{i \in ss} p_i$ ensures that we only consider rules for which some component in ss is involved: if that is not the case, the rule cannot trigger a subsystem transition as we discuss in Sect. 3. Unlike the other sub-formulæ, *LocallyBlocked* is not in CNF. There is, however, a well-known transformation that, for any input boolean formula, creates an equisatisfiable CNF formula in polynomial time [Tse68].

We use the triple-disjoint assumption to create a more compact encoding for this predicate. Therefore, this encoding, and consequently our implementation of *Pair*, can only be soundly applied to triple-disjoint systems. It should be noted, however, that a blocked constraint that does not rely on triple-disjointness could be constructed in polynomial time by encoding how components can trigger (participate on) system rules.

This translation only takes polynomial time as it only requires the explicit analysis of subsystems of size at most 2. Moreover, since each sub-formula captures its corresponding counterpart in our *Pair*-(local-)candidate definition, our encoding soundly captures *Pair*-(local-)candidate detection. Therefore, a satisfying assignment found by these formulæ gives rise to a valid (corresponding) *Pair* (local) candidate, whereas unsatisfiability implies *Pair*-(local-)candidate freedom and, in turn, (local-)deadlock freedom.

5.4. Practical evaluation

In this section, we evaluate our framework, which is implemented in our *DeadlOx* tool. This tool and the models used in this section are available at [AGRR18] It uses output from FDR4 to generate our SAT encoding which is then checked by the Glucose 4.0 solver [AS09]. FDR4 is used as a library to translate/compile CSP models into supercombinator machines. We use CSP as an input language because we believe it provides concise and simple descriptions for concurrent systems. However, any other language could be used if a translation into supercombinator machines is provided. We extend the input language of FDR4 with annotation `:[Pair]` that should be added to a (local-)deadlock free assertion; it tells FDR4 to use our *Pair* technique instead of explicit state exploration to check the assertion. We added the assertion `:[sublock free [F]]`, which checks for local-deadlock freedom and can only be used with the Pair annotation. For instance, a distributed system described by process `SYSTEM` could be checked using *Pair* by the following assertions.

```
assert SYSTEM :[deadlock free [F]] :[Pair]
assert SYSTEM :[sublock free [F]] :[Pair]
```

Our experiment evaluates deadlock and local-deadlock freedom for some triple-disjoint deadlock-free systems. The experiment was conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. We compare DeadlOx deadlock-checking (Pd) and local-deadlock-checking (Pl) using *Pair* against the Deadlock Checker [MJ97], FDR4's deadlock freedom assertion (FDR) [GRABR14], and D-Finder 2 [BGL+11]. Deadlock Checker implements the SDD framework, FDR4 is a complete/exact method with explicit space exploration, and D-Finder 2 is an approximate approach that uses some tailored system invariants. D-Finder 2 implements three techniques to calculate these invariants: a boolean-constraint-based (DF2pm), a fixed-point-based (DF2fp), and an enumerative one (DF2l). Also, when appropriate, we combine FDR4's explicit state exploration with partial order reduction (FDRp) [GRHRW15] or compression techniques (FDRc) [RGG+95]. While SDD proves a property that is stronger than local-deadlock freedom, D-Finder 2 and FDR4 methods check only for deadlock freedom. We know of no reasonable approach to checking local-deadlock freedom with these tools. In fact, hereditary deadlock freedom implies local-deadlock freedom, but we know of no reasonable way to prove hereditary deadlock freedom either.

We chose eleven benchmark systems that are proved local-deadlock free by *Pair*. These systems implement the alternating bit protocol (ABP), the asymmetric solution to the dining philosophers (Phils) and three versions of the well-known butler solution (ButI, ButID, ButID2), a grid network implementing Tarry's algorithm (Tarry) [Tar95], a central lock system (Lock), and a grid network implementing a simplified implementation of Raymond's algorithm (Ray) [Ray89], a binary telephone switch (Tel), the mad postman routing algorithm (Rout) [YJ89, Ros98], the sliding window protocol (SWP). Most of these models are introduced and discussed in [Ros10]. Tarry's algorithm finds a spanning tree over the communication graph of a distributed system, and Raymond's algorithm is used to achieve mutual exclusion.

Table 1 presents the results for the hereditary deadlock-free systems. As for approximative methods, these results suggest that our method scales similarly to SDD. *Pair* can show (local-)deadlock freedom for both Tel and Tarry examples while SDD cannot. This demonstrates what we informally claimed when we compared *Pair* and SDD, namely, *Pair* is better than SDD even for hereditary deadlock-free systems. D-Finder 2's approaches, although approximate, seem to be much less efficient than any other method, even complete ones. It seems that the calculation of invariants this tool carries out is rather complex for these examples. Also, it might be the case that our generation of BIP models (the input language for D-Finder 2) from supercombinator machines does not provide an optimal encoding for BIP systems.

As for exact methods, *Pair* fares better than FDR4's techniques for most examples. For Lock, Tarry and SWP, FDR4's explicit exploration, however, fares better. For these examples, the system's state space does not grow so rapidly as the number of components increases, hence, FDR4's good performance. For Tarry and SWP, the state space of individual components grows rather drastically with the increase of N. *Pair*'s quadratic increase in analysing pairs of components combined with this rapid growth is the reason behind *Pair*'s lack of scalability. However, the combination of FDR4's deadlock-free assertion with compression techniques can be remarkably efficient for some systems. The use of compression, however, requires manually devising a compression strategy, whereas all other methods are fully automatic. Also, a lot of skill is necessary in devising effective strategies.

For the Tarry example, our tool only manages to prove local-deadlock freedom for $N = 5$. The components in this system are arranged in a $5\times(N/5)$ grid that uses a token mechanism to construct a spanning tree for this grid; there invariant that a token always exists in the system is fundamental to prove deadlock and local-deadlock freedom. For $N = 5$, we have a 5×1-grid network where components can only communicate with their left and right neighbours, whereas for $N = 10$, $N = 12$ and $N = 15$, we have a grid-like systems where components can communicate additionally with up and down neighbours. For $N = 5$, the pairwise analysis (carried out by $reach_2$) can keep track of the token and show that subsystems are never blocked. For the other instances, however, the routes the the token might take around the network are too unpredictable and so pairwise analysis cannot capture the required token invariant.[2]

Table 2 presents the results for the non-hereditary deadlock-free systems. SDD (and any framework based on the detection of cycles of ungranted requests) is unable to show (local-)deadlock freedom for systems in this class. On the other hand, *Pair* can prove (local-)deadlock freedom for some systems in this class. It can show (local-)deadlock freedom for these three variants of the butler solution to the dining philosophers problem. ButI is a solution with a single butler that allows only $N - 1$ philosophers to sit at the table, where N is the total number of philosophers, and it does that by keeping track of the identity of philosophers sat at the table. Hence, the state space of this butler component grows exponentially as N increases. This growth is the reason why *Pair* does not scale for this example. ButID circumvents this problem by having $N - 1$ butlers each of which allow a philosopher to the table. *Pair* does not scale well for this example either. As N grows, the size of individual components increases linearly, the number of components increase linearly, and the number of egdes in the communication graph grows quadratically. So, *Pair*'s lack of scalability comes from the fact that it needs to analyse a quadratic number of pairs of components and each of these analyses creates a constraint that is quadratic in the size of the components. Finally, ButID2 creates a solution that is more amenable to *Pair*. In this solution, we have $N - 1$ butlers but each of them only takes care of 5 fixed philosophers. In this setting, as N grows, the size of components remains constant, the number of components increases linearly, and the number of connections in the communication graph grows linearly.

These solutions are different from the more traditional one where a butler simply counts the number of philosophers sat at the table regardless of their identity. This traditional solution, however, cannot be proved deadlock free by *Pair* as it requires global analysis of the system. Intuitively, by adding the identity of philosophers we transform the global invariant "a philosopher must be left out of the table at all times" into a pairwise one that *Pair* can capture. To illustrate this difference let us assume we have a system with 3 philosophers and 3 forks. In the counting-butler case, each pair butler-philosopher can reach the state where the butler has counted until 2 and the philosopher is sat at the table; this philosopher and another one could have sat at the table. Hence, 2-reachability cannot discharge (i.e. prove unreachable) the system state where the butler has counted until 2 and all philosophers are sat at the table simultaneously. In the case where the butler identifies philosophers, however, 2-reachability does discharge system states where all philosophers are sat at the table simultaneously. We represent butler states using b_S where S is a set representing the philosophers sat at the table; we have that $S \subset \{0, 1, 2\}$, and 0, 1, and 2 are the identifiers of our philosophers. From the analysis of the pair composed of the butler and philosopher i where $i \in \{0, 1, 2\}$, we can capture that philosopher i cannot be sat at the table when the butler is in a state S such that $i \notin S$. Thus, from the analysis of these pairs, 2-reachability can deduce that the only way all philosophers can sit at the table simultaneously is if the butler reaches state $\{0, 1, 2\}$. This state, however, is not part of the butler's state space, since it is exactly the state it is trying to prevent. Example 3 gives a more detailed account of the dining-philosophers system with a butler that identifies philosophers.

[2] We present in other works techniques that specifically captures this sort of token-based invariants.

Table 1. Results for hereditary deadlock-free systems

		Approximate						Exact		
Example	N	SDD	Pl	Pd	DF2pm	DF2fp	DF2l	FDR	FDRc	FDRp
ABP	10	0.15	0.06	0.06	*	276.52	*	0.46	\	0.11
	30	0.23	0.12	0.06	*	*	*	0.16	\	0.16
	50	0.38	0.11	0.11	*	*	*	0.31	\	0.16
	70	0.53	0.17	0.16	*	*	*	1.17	\	0.21
Lock	50	0.23	0.78	0.11	*	*	*	0.11	\	0.11
	100	0.33	0.22	0.21	*	*	*	0.11	\	0.41
	200	0.68	0.67	0.62	*	*	*	0.16	\	3.67
	500	5.04	4.82	4.77	*	*	*	0.31	\	100.50
Phils	50	0.28	0.11	0.12	*	*	*	*	1.47	0.52
	100	0.38	0.22	0.16	*	*	*	*	13.29	5.47
	200	0.53	0.37	0.31	*	*	*	*	219.82	62.55
	500	1.23	1.07	0.82	*	*	*	*	*	*
Ray	10	0.48	0.06	0.26	*	*	*	0.12	\	0.11
	25	0.20	0.12	0.11	*	*	*	49.65	\	18.39
	50	0.23	0.22	0.11	*	*	*	*	\	*
	100	0.33	0.32	0.21	*	*	*	*	\	*
SWP	3	1.23	0.21	0.67	*	*	*	0.41	0.27	0.51
	4	38.21	2.17	2.02	*	*	*	2.87	1.07	6.22
	5	*	145.23	57.44	*	*	*	47.51	4.02	97.14
	6	*	*	*	*	*	*	*	21.04	*
Tarry	5	–	0.22	0.12	3.87	6.83	5.82	0.06	\	0.06
	8	–	–	0.11	*	*	*	0.11	\	0.11
	12	–	–	26.92	*	*	*	0.41	\	0.57
	15	–	–	*	*	*	*	39.30	\	48.61
Tel	4	–	0.11	0.11	*	*	*	*	3.17	*
	6	–	0.42	0.31	*	*	*	*	*	*
	8	–	3.67	2.68	*	*	*	*	*	*
	10	–	87.89	46.84	*	*	*	*	*	*
Rout	5	0.86	0.11	0.57	*	44.20	*	*	0.26	*
	10	0.38	0.27	0.21	*	*	*	*	0.71	*
	20	0.98	0.92	0.82	*	*	*	*	4.47	*
	30	2.13	3.07	2.37	*	*	*	*	16.25	*

N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to check (local-)deadlock freedom for each system
The symbol * means that the method took longer than 300 s, or some error occurred, such as running out of memory
The symbol – means that the method is unable to prove (local-)deadlock freedom
The symbol \ means that no efficient compression technique could be found

Unsurprisingly, *Pair* is not able to prove (local-)deadlock freedom when this property depends on some global invariant preserved by the system (or perhaps by larger subsets of the system than the pairs used here). For instance, proving (local-)deadlock freedom for Milner's scheduler, which is a fairly simple well-known system, is out of our method's reach. The issue with Milner's scheduler is that it is essentially a token ring for which (local-)deadlock freedom 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. All the examples we tried D-Finder 2 on were disappointing. Last but not least we want to highlight the somewhat surprising fact that checking local-deadlock freedom and deadlock freedom via *Pair* scale similarly. One would perhaps expect local-deadlock freedom to be a lot harder to check as it involves some (seemingly difficult) quantification over all subsystems of the system under analysis. It seems, however, that the sort of existential quantification over our participation variables is no trouble for modern SAT solvers.

6. PairPicking: *Pair* meets user's picks

In this section, we introduce the *PairPicking*, a simple strategy that is meant to address some of *Pair*'s imprecision at no substantial cost regarding scalability. This strategy is not a different framework in itself but a different way in which *Pair* can be sharpened by some user input to tackle some of its imprecision.

Table 2. Results for non-hereditary deadlock-free systems

		Approximate						Exact		
Example	N	SDD	Pl	Pd	DF2pm	DF2fp	DF2l	FDR	FDRc	FDRp
ButID	3	–	0.06	0.11	6.38	*	9.03	0.11	0.11	0.11
	5	–	0.17	0.11	*	*	*	0.26	0.67	0.16
	7	–	0.22	0.16	*	*	*	284.81	40.74	15.14
	10	–	66.14	71.00	*	*	*	*	*	*
ButID2	3	–	0.06	0.46	13.19	*	36.58	0.11	0.17	0.11
	5	–	0.17	0.11	*	*	*	0.31	0.67	0.16
	10	–	0.22	0.11	*	*	*	*	*	*
	50	–	0.82	0.62	*	*	*	*	*	*
ButI	3	–	0.11	0.51	4.87	60.99	5.47	0.11	0.11	0.11
	5	–	0.12	0.06	*	*	*	0.11	0.16	0.11
	10	–	0.52	0.41	*	*	*	127.39	0.41	0.62
	15	*	*	*	*	*	*	*	14.59	46.19

N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to check deadlock freedom for each system
The symbol * means that the method took longer than 300 s, or some error occurred, such as running out of memory
The symbol – means that the method is unable to prove deadlock freedom

Pair is a fully automatic framework designed around the analysis of pairs of components, and, as such, it cannot prove (local-)deadlock freedom when this property depends on the behaviour of triples or larger combinations of components. In PairPicking, we try to tackle this imprecision by giving the user the ability to manually identify some subsystems, involving more than two components, they believe are necessary in proving (local-)deadlock freedom. The reachability approximation derived from these subsystems are combined with $reach_2$. Unlike *Pair*, we do not calculate the reachability approximation for a subsystem and lift it to the entire system. Instead, we replace the chosen subsystems by their projections. We rely on the following supercombinator machine to carry out this substitution. Note that the following definition requires rules to have distinct system events. A system can be converted to this format by changing rules with the same system events, so they all have distinct ones. This change does not introduce or remove deadlock states of the original system, and it can be carried out efficiently by simply iterating over the system's rules.

Definition 19 Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where rules in $\mathcal{R}$ have *distinct* system events (i.e. if (e, a) and (e', a) are members of $\mathcal{R}$ then $e = e'$), and subsystems $ss_1, \ldots, ss_m$ partitioning the system, i.e. where $\bigcup_{i \in \{1 \ldots m\}} ss_i = \{1 \ldots n\}$ and $ss_i \cap ss_j = \emptyset$ for $i \neq j$ hold. $\mathcal{S}_{ss_1,\ldots,ss_m}$ is the supercombinator machine where the combination of components in each subsystem ss_i is replaced by its projection's LTS.

$$\mathcal{S}_{ss_1,\ldots,ss_m} = (\langle L_{ss_1}, \ldots, L_{ss_m} \rangle, \mathcal{R}') \text{ where}$$

- L_{ss} the LTS induced by $\mathcal{S}_{ss}$ as per Definition 9.
- $\mathcal{R}' = \{(\{(i, a) \mid i \in \{1 \ldots m\} \wedge \exists j \in ss_i \bullet e_j \neq -\}, a) \mid ((e_1, \ldots, e_n), a) \in \mathcal{R}\}$

Since we also make sure that rules are adapted to enforce the same interactions as the original ones, this supercombinator machine has the same behaviour as the original one. So, it (local-)deadlocks whenever the original does and, thus, we can freely replace a supercombinator machine by this modified version when checking for (local-)deadlock freedom. In particular, we can apply *Pair* to this modified machine instead; this application is exactly what constitutes the PairPicking strategy.

Theorem 6 Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where rules in $\mathcal{R}$ have *distinct* system events, and $ss_1, \ldots, ss_m$ a set of subsystems such that they partition the system. $\mathcal{S}_{ss_1,\ldots,ss_m}$ has a (local-)deadlock iff $\mathcal{S}$ does.

Proof This follows from our definition of $\mathcal{S}_{ss_1,\ldots,ss_m}$ using induction on the size of paths leading to a (local-) deadlock. QED □

This modified version of a supercombinator machine does not alter its behaviour but it does alter its structure. When applied to this machine, *Pair* would consider, as this machine does, the parallel combination of components in each subsystem as individual components. By analysing the overall behaviour of these combinations, *Pair* is

implicitly using reachability approximations induced by these subsystems. Therefore, the application of *Pair* to such a modified machine where triples or larger combination of components are put together creates a framework that precisely understands how these larger subsystems work and, thus, is more precise in showing (local-)deadlock freedom.

In some cases when *Pair* fails to prove (local-)deadlock freedom for a system, it is easy to understand why it fails and to recognise which triples or larger subsystems are additionally needed to prove this property. If triples or larger combinations are reasonably small, we have the perfect setting to apply PairPicking. The user of this strategy can create the proposed modified machine with these combinations and *Pair* will be able to verify it.

The complexity of this strategy can be as bad as explicit state space analysis of the entire system; after all we do not forbid the user from choosing the entire system as a subsystem. So, given the size of the original machine and the chosen subsystem, the problem of checking (local-)deadlock freedom using the PairPicking strategy with unlimited subsystem size should also be *PSPACE*-complete. Nevertheless, it should be clear that by picking small subsystems, this strategy is only explicitly analysing small state spaces. Hence, in practice, if small subsystems are chosen, this strategy should considerably outperform explicit state-space exploration of the overall system.

Now, we introduce two families of systems and discuss how to apply PairPicking to them. We apply the PairPicking strategy using our implementation of *Pair* and the built-in function `explicate` of FDR4. This function constructs the induced LTS for a given subsystem. So, applying PairPicking involves using `explicate` to create the modified system and *Pair* to check it for (local-)deadlock freedom.

These families of systems describe routing networks where messages are exchanged between small local networks. Each of these local networks is composed of a fixed number of components which initially decide on a single *interface* component for this local network. Local networks exchange messages only through their interface components. Our two families of systems differ in the way local networks choose their interface component.

In the first family, a local network elects an interface component using a majority vote. Each component in the local network has a single vote and they vote on each other until a component receives the majority of votes, becoming the interface component. In the second family, components in a local network elects an interface component based on a priority value they choose. Each component chooses a priority values that is sent around the network. The component with the highest priority, where the component unique identifier is a tie-breaker, is the elected interface component [Tel00].

Proving (local-)deadlock freedom for these routing systems rests on, among other invariants, the fact they succeed in choosing an interface component. This invariant, however, cannot be captured by *Pair* alone for either of these families. On the other hand, PairPicking can show (local-)deadlock freedom for these systems if we treat local networks as individual components.

We applied PairPicking for systems in these two families; we replaced local networks by their induced LTSs. For each family, we vary the number of components in a local network and the topology of connections between local networks: they can be laid out as a chain, a grid or a fully-connected graph. This experiment was conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. We compare PairPicking checking deadlock freedom (PPd) and local-deadlock freedom (PPl) against FDR4's approaches [GRABR14]. *Pair* and the Deadlock Checker's SDD [MJ97] are left out because they cannot prove any of these systems (local-)deadlock free. D-Finder 2 [BGL+11]'s approaches are also omitted because they either timeout or cannot prove deadlock freedom for all these systems. FDR4 methods only check for deadlock freedom.

Table 3 presents the results for the voting-based family of systems, whereas Table 4 presents the results for systems in the priority-based family. The name of each example describes the topology used to connect local networks and the number of components in each of them. For instance, VGrid4 is the system where local networks are connected in a grid-like fashion and each of them is composed of 4 components. These results show that PairPicking is quicker in showing (local-)deadlock freedom for these systems than complete approaches. FDR4's assertion combined with compression techniques comes close to the sort of speed

PairPicking achieves. In some cases, however, the sort of manual work needed to use PairPicking, namely, selecting the subsystems that need to be explicit examined, is much less complex than the work needed to craft a compression strategy.

These results also confirm the practical limitations of PairPicking. Since it performs explicit exploration to create the LTSs of the chosen subsystems, it suffers with the state-space explosion problem. So, even for small subsystems, the size of their LTSs tends to make the verification cost prohibitive—even for approximative approaches such as *Pair*. Furthermore, these results also demonstrate how the topology affects the underlying use of *Pair*. A system with a fully-connected topology normally suffers from two sources of complexity as the number of components grows. Firstly, with the addition of a component, there is an increase in the complexity of individual components as they have to account for the communication with this newly included component. Secondly, the number of connections between components, and of the pairs of components to explicitly analyse, grows quadratically with the linear increase of components. On the other hand, for communication topologies where each component is connected to a fixed number of components, such as grids, rings or chains, neither of these two problems arise. These factors help to explain the difference in scalability for the different systems (and topologies) we have analysed. One could presumably use *Pair* to guide PairPicking in the sense that, when a *Pair* candidate arises, one could examine which components prevent it from being a truly reachable state, if that is the case, and input the subsystem involving them to PairPicking.

7. Conclusion

This paper's main object of study is local analysis. We propose a way to capture and implement it, and we show how it can enable effective verification frameworks.

We propose the notion of subsystem reachability as a means to capture and implement local analysis. It can be used in its own right to approximate reachability but a question that arises is which subsystems one should use to construct such an approximation. There is no easy answer to this question. It is fairly difficult to anticipate which subsystem plays a role in enforcing a given property. Also, it might be the case that a property emerges from the behaviour of not one but many small subsystems. To alleviate these problems, we propose the notion of k-reachability. In a straight-forward way, it picks subsystems of size(-up-to) k to construct a reachability over-approximation. These notions are in no way tied to a particular property such that they could be applied to any verification that could be reduced to a reachability check.

We use 2-reachability to create *Pair*, an approximate framework that checks deadlock and local-deadlock freedom. The use of this approximation tackles some sources of imprecision of traditional techniques that check for cycles of dependencies. It improves the accuracy of current approximate techniques; in particular, some non-hereditary deadlock-free systems, which are neglected by most approximate techniques, can be tackled by our framework. This improvement comes at a price. The problems tackled by *Pair* are *co-NP*-hard, whereas traditional approaches rely on conditions that can be checked in polynomial time. Still, *Pair*-(local-)candidate detection should be easier to handle if compared to the *PSPACE*-completeness of exact (local-)deadlock checking. Intuitively, *Pair* needs to analyse pairs of components to approximate reachability whereas exact frameworks need to go over the system's entire state space. Despite the inherent complexity of detecting *Pair* (local) candidates, SAT checkers can efficiently implement our framework. Our implementation, which can only handle triple-disjoint systems, is in most cases similarly efficient as traditional approximate techniques and much better than exact frameworks. This is demonstrated by a series of practical experiments. Moreover, these solvers can even handle the sort of (seemingly costly) quantification over subsystems that is necessary to show local-deadlock freedom.

We have not investigated the use of k-reachability where $k > 2$. Most of the examples we have worked with were either proved (local-)deadlock free by local invariants calculated by pairwise analysis or by global invariants. Also, we could not immediately think of a relevant class of systems that would require this sort of k-reachability. Obviously, that does not mean such a class does not exist. Moreover, we have not investigated how our reachability approximation fares in showing (local-)deadlock freedom for non-triple-disjoint systems. We hope that this paper will encourage further work in this direction.

Pair cannot show (local-)deadlock freedom when it depends on some reachability invariant of triples or larger combinations of components. To cope with that, we propose PairPicking, a strategy that combines the reachability analysis of some hand-picked subsystems with *Pair*. The choosing of subsystems enables the user to capture invariants of triples or larger combinations of components.

Table 3. Efficiency comparison for networks of voting-based local subnetworks

		Approximate		Exact		
Example	N	PPl	PPd	FDR	FDRc	FDRp
VChain4	10	0.36	0.47	*	1.52	*
	20	0.52	0.46	*	5.98	*
	30	0.72	0.66	*	19.95	*
	40	0.97	0.87	*	56.63	*
VGrid4	10	0.41	0.52	*	*	*
	20	1.17	1.27	*	*	*
	30	1.92	2.22	*	*	*
	40	2.62	2.62	*	*	*
VFully4	5	0.42	0.61	*	*	24.30
	8	2.27	2.02	*	*	*
	12	9.99	8.73	*	*	*
	15	22.46	21.56	*	*	*
VChain5	10	1.02	1.22	*	27.92	*
	20	2.12	2.07	*	69.40	*
	30	3.32	3.12	*	135.98	*
	40	4.47	4.27	*	266.88	*
VGrid5	10	1.92	1.82	*	*	*
	20	5.48	5.58	*	*	*
	30	9.03	8.98	*	*	*
	40	12.99	12.39	*	*	*
VFully5	5	1.82	2.02	*	*	118.33
	8	8.98	8.53	*	*	*
	12	41.70	39.29	*	*	*
	15	105.48	97.15	*	*	*
VChain6	10	14.49	15.19	*	*	*
	20	31.43	31.78	*	*	*
	30	49.06	48.86	*	*	*
	40	66.99	65.74	*	*	*
VGrid6	10	25.66	25.61	*	*	*
	20	67.00	67.69	*	*	*
	30	110.44	110.18	*	*	*
	40	153.23	150.36	*	*	*
VFully6	5	21.36	20.70	*	*	*
	8	89.89	85.03	*	*	*
	12	289.84	282.70	*	*	*
	15	*	*	*	*	*

N gives the number of local networks in the system. We measure in seconds the time taken to check (local-)deadlock freedom for each system
The symbol * means that the method took longer than 300 s, or an error, such as running out of memory, occurred
The symbol – means that the method is unable to prove deadlock freedom

This strategy should be applied when these subsystems are small and easy to identify. Some works have proposed the techniques to find global reachability invariants [Mar96, CK94, DCCN04, AGRR16b, AGRR17b]. In [AGRR16b, AGRR17b, Ant18], we propose some techniques to estimate global reachability and a framework that integrates local analysis (in the form of 2-reachability) and these global-analysis techniques to verify systems.

Local analysis is a tool that can prove properties of systems emerging from small combination of components. Hence, the frameworks proposed in this paper should not prove (local-)deadlock freedom if it depends on some global invariant of the system. Nevertheless, our use of local analysis should make our frameworks, generally, much quicker than exact techniques for verifying systems. So, they could be used as preliminary test for (local-)deadlock freedom. Furthermore, despite being imprecise in the negative case, they still present a candidate (local-)deadlock. Although it is not as useful as a true counter-example, this candidate can provide some insight as to whether the system has a true (local-)deadlock or not. In some cases, it might be evident that the candidate is actually reachable, and consequently, a real violation.

Table 4. Efficiency comparison for networks of priority-based local subnetworks

		Approximate		Exact		
Example	N	PPl	PPd	FDR	FDRc	FDRp
PChain4	5	0.57	0.87	*	0.92	5.18
	10	1.17	1.12	*	2.07	*
	15	1.77	1.72	*	3.27	*
	20	2.47	2.37	*	6.28	*
PGrid4	5	0.57	0.51	*	0.87	5.12
	10	2.17	2.02	*	*	*
	15	3.82	4.07	*	*	*
	20	5.43	5.83	*	*	*
PFully4	3	0.46	0.67	18.95	1.87	0.22
	5	1.67	1.62	*	*	44.44
	8	7.03	6.88	*	*	*
	12	26.87	25.47	*	*	*
PChain5	5	4.62	4.87	*	7.73	35.47
	10	10.69	10.74	*	18.05	*
	15	16.75	16.80	*	29.62	*
	20	22.96	22.91	*	44.25	*
PGrid5	5	4.67	4.72	*	7.68	35.52
	10	18.05	18.25	*	*	*
	15	32.88	32.98	*	*	*
	20	48.71	48.86	*	*	*
PFully5	3	3.57	3.72	*	41.19	0.71
	5	14.64	14.54	*	*	254.25
	8	56.38	53.26	*	*	*
	12	170.85	165.38	*	*	*
PChain6	5	46.56	47.55	*	278.26	234.85
	10	106.78	107.92	*	*	*
	15	167.39	169.48	*	*	*
	20	229.36	229.17	*	*	*
PGrid6	5	46.46	47.30	*	259.57	235.38
	10	173.76	173.46	*	*	*
	15	*	*	*	*	*
	20	*	*	*	*	*
PFully6	3	34.88	35.99	*	*	3.90
	5	130.22	128.18	*	*	*
	8	*	*	*	*	*
	12	*	*	*	*	*

N gives the number of local networks in the system. We measure in seconds the time taken to check (local-)deadlock freedom for each system
The symbol * means that the method took longer than 300 s, or an error, such as running out of memory, occurred
The symbol – means that the method is unable to prove deadlock freedom

As future work, we plan to investigate a few points. *Pair*'s analysis simply enumerates pairs of components states that cannot be reached. So, there may be a quadratic increase in the number of pairs to be analysed and in their state space, leading to a substantial increase in the constraint we feed to the SAT solver. In some cases, this increase hinders the efficiency of solvers. Therefore, we plan to investigate ways in which we can reduce the size of this constraint. Perhaps, we could analyse fewer pairs of components and use a BDD representation to reduce the final boolean constraint. In PairPicking, one avenue that we have not explored is the use of compression techniques in the construction of LTSs of hand-picked subsystems. These techniques could greatly reduce these LTSs making PairPicking much more scalable.

Acknowledgements

The first author is a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) Foundation scholarship holder (Process No: 13201/13-1). The second and third authors are partially sponsored by EPSRC (Engineering and Physical Sciences Research Council, UK) under Agreement No. EP/N022777, and

Innovate UK and the Aerospace Technology Institute via the SECT-AIR Project under Agreement No. 113099. We thank the anonymous reviewers for their valuable comments that helped improve this paper.

References

[ABB+13] Attie PC, Bensalem S, Bozga M, Jaber M, Sifakis J, Zaraket FA (2013) An abstract framework for deadlock prevention in BIP. In: FORTE, number 7892 in LNCS. Springer, pp 161–177

[ABB+18] Attie PC, Bensalem S, Bozga M, Jaber M, Sifakis J, Zaraket FA (2018) Global and local deadlock freedom in BIP. ACM Trans Softw Eng Methodol 26(3):9:1–9:48

[ABC+91] Avrunin GS, Buy UA, Corbett JC, Dillon LK, Wileden JC (1991) Automated analysis of concurrent systems with the constrained expression toolset. IEEE Trans Softw Eng 17(11):1204–1222

[AC05] Attie PC, Chockler H (2005) Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In: VMCAI. Springer, pp 465–481

[AFDR80] Apt KR, Francez N, De Roever WP (1980) A proof system for communicating sequential processes. ACM Trans Program Lang Syst (TOPLAS) 2(3):359–385

[AGRR16a] Antonino P, Gibson-Robinson T, Roscoe AW (2016) Efficient deadlock-freedom checking using local analysis and SAT solving. In: IFM, number 9681 in LNCS. Springer, pp 345–360

[AGRR16b] Antonino P, Gibson-Robinson T, Roscoe AW (2016) Tighter reachability criteria for deadlock freedom analysis. In: FM, number 9995 in LNCS. Springer

[AGRR17a] Antonino P, Gibson-Robinson T, Roscoe AW (2017) The automatic detection of token structures and invariants using SAT checking. In: TACAS, number 10206 in LNCS. Springer, pp 249–265

[AGRR17b] Antonino P, Gibson-Robinson T, Roscoe AW (2017) Checking static properties using conservative sat approximations for reachability. In: Formal methods: foundations and applications. Springer, pp 233–250

[AGRR18] Antonino P, Gibson-Robinson T, Roscoe AW (2018) Experiment package. www.cs.ox.ac.uk/people/pedro.antonino/facpkg.zip

[Ant18] Antonino P (2018) Verifying concurrent systems by approximations. DPhil thesis, University of Oxford. https://ora.ox.ac.uk/objects/uuid:f75c782c-a168-49b3-bfed-e2715f027157

[AOS+14] Antonino P, Oliveira MM, Sampaio A, Kristensen K, Bryans J (2014) Leadership election: an industrial SoS application of compositional deadlock verification. In: NFM, volume 8430 of LNCS, pp 31–45

[AS09] Audemard G, Simon L (2009) Predicting learnt clauses quality in modern SAT solvers. In: IJCAI'09, San Francisco, CA, USA, pp 399–404.

[ASW14] Antonino P, Sampaio A, Woodcock J (2014) A refinement based strategy for local deadlock analysis of networks of CSP processes. In: FM, volume 8442 of LNCS, pp 62–77

[BBL+16] Bensalem S, Bozga M, Legay A, Nguyen T-H, Sifakis J, Yan R (2016) Component-based verification using incremental design and invariants. Softw Syst Model 15(2):427–451

[BCCZ99] Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without bdds. In: Tools and algorithms for the construction and analysis of systems, pp 193–207

[BCM+92] Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang L-J (1992) Symbolic model checking: 1020 states and beyond. Inf Comput 98(2):142–170

[BGL+11] Bensalem S, Griesmayer A, Legay A, Nguyen T-H, Sifakis J, Yan R (2011) D-finder 2: towards efficient correctness of incremental design. In: NFM, pp 453–458

[BK91] Barghouti NS, Kaiser GE (1991) Concurrency control in advanced database applications. ACM Comput Surv 23(3):269–317

[BK08] Baier C, Katoen J-P (2008) Principles of model checking (representation and mind series). The MIT Press

[BL99] Bensalem S, Lakhnech Y (1999) Automatic generation of invariants. Form Methods Syst Des 15(1):75–92

[BR91] Brookes SD, Roscoe AW (1991) Deadlock analysis in networks of communicating processes. Distrib Comput 4:209–230

[CA95] Corbett JC, Avrunin GS (1995) Using integer programming to verify general safety and liveness properties. Form Methods Syst Des 6(1):97–123

[CCO+05] Chaki S, Clarke E, Ouaknine J, Sharygina N, Sinha N (2005) Concurrent software verification with states, events, and deadlocks. Form Asp Comput 17(4):461–483

[CES71] Coffman EG, Elphick M, Shoshani A (1971) System deadlocks. ACM Comput Surv (CSUR) 3(2):67–78

[CGJ+00] Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Computer aided verification. Springer, pp 154–169

[CK94] Cheung SC, Kramer J (1994) Tractable dataflow analysis for distributed systems. IEEE Trans Softw Eng 20(8):579–593

[Cor96] Corbett JC (1996) Evaluating deadlock detection methods for concurrent software. IEEE Trans Softw Eng 22(3):161–180

[DCCN04] Dwyer MB, Clarke LA, Cobleigh JM, Naumovich G (2004) Flow analysis for verifying properties of concurrent software systems. ACM Trans Softw Eng Methodol 13(4):359–430

[FOSC16] Conserva Filho MS, Oliveira MVM, Sampaio A, Cavalcanti A (2016) Local livelock analysis of component-based models. In: ICFEM, pp 279–295
[GRABR14] Gibson-Robinson T, Armstrong P, Boulgakov A, Roscoe AW (2014) FDR3—a modern refinement checker for CSP. In: TACAS, volume 8413 of LNCS, pp 187–201
[GRHRW15] Gibson-Robinson T, Hansen H, Roscoe AW, Wang Xu (2015) Practical partial order reduction for CSP. In: NFM, volume 9058 of LNCS. Springer, pp 188–203
[GW93] Godefroid P, Wolper P (1993) Using partial orders for the efficient verification of deadlock freedom and safety properties. FMSD, 2(2):149–164
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Upper Saddle River
[HS08] Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann Publishers Inc., San Francisco
[JL16] Jezequel L, Lime D (2016) Lazy reachability analysis in distributed systems. In: Desharnais J, Jagadeesan R (eds) CONCUR 2016, volume 59 of Leibniz international proceedings in informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2016, pp 17:1–17:14
[KS90] Kanellakis PC, Smolka SA (1990) Ccs expressions, finite state processes, and three problems of equivalence. Inf Comput 86(1):43–68
[Lam77] Lamport L (1977) Proving the correctness of multiprocess programs. IEEE Trans Softw Eng (2):125–143
[LMC11] Lambertz C, Majster-Cederbaum M (2011) Analyzing component-based systems on the basis of architectural constraints. In: FSEN. Springer, pp 64–79
[Mar96] Martin Jeremy MR (1996) The design and construction of deadlock-free concurrent systems. Ph.D. thesis, University of Buckingham
[MJ97] Martin JMR, Jassim SA (1997) An efficient technique for deadlock analysis of large scale process networks. In: FME '97, pp 418–441
[OAR+16] Oliveira MVM, Antonino P, Ramos R, Sampaio A, Mota A, Roscoe AW (2016) Rigorous development of component-based systems using component metadata and patterns. Form Asp Comput 1–68
[OCS17] Otoni R, Cavalcanti A, Sampaio A (2017) Local analysis of determinism for CSP. In: Proceedings of formal methods: foundations and applications—20th Brazilian symposium, SBMF 2017, Recife, Brazil, 29 November–1 December 2017, pp 107–124
[OPRW13] Ouaknine J, Palikareva H, Roscoe AW, Worrell J (2013) A static analysis framework for livelock freedom in CSP. LMCS, 9(3)
[Pel93] Peled D (1993) All from one, one for all: on model checking using representatives. In: Computer aided verification. Springer, pp 409–423
[Plo81] Plotkin GD (1981) A structural approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Department, Aarhus University
[POR12] Palikareva H, Ouaknine J, Roscoe AW (2012) SAT-solving in CSP trace refinement. Sci Comput Program 77(10):1178–1197
[Ram11] Ramos RT (2011) Systematic development of trustworthy component-based systems. Ph.D. thesis, Universidade Federal de Pernambuco
[Ray89] Raymond K (1989) A tree-based algorithm for distributed mutual exclusion. ACM Trans Comput Syst (TOCS) 7(1):61–77
[RD87] Roscoe AW, Dathi N (1987) The pursuit of deadlock freedom. Inf Comput 75(3):289–327
[RGG+95] Roscoe AW, Gardiner PHB, Goldsmith M, Hulance JR, Jackson DM, Scattergood JB (1995) Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In: TACAS, pp 133–152
[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice Hall, Upper Saddle River
[Ros10] Roscoe AW (2010) Understanding Concurrent Systems. Springer
[Sav70] Savitch WJ (1970) Relationships between nondeterministic and deterministic tape complexities. J Comput Syst Sci 4(2):177–192
[SD82] Scholten CS, Dijkstra EW (1982) A class of simple communication patterns. Springer, New York, pp 334–337
[Tar95] Tarry G (1895) Le probleme des labyrinthes. Nouvelles annales de mathématiques. journal des candidats aux écoles polytechnique et normale 14:187–190
[Tel00] Tel G (2000) Introduction to distributed algorithms, 2nd edn. Cambridge University Press, Cambridge
[TGS17] Timm N, Gruner S, Sibanda P (2017) Model checking of concurrent software systems via heuristic-guided sat solving. In: Dastani M, Sirjani M (eds) Fundamentals of software engineering. Springer, Cham, pp 244–259
[Tse68] Tseitin G (1968) On the complexity of derivation in propositional calculus. Stud Constrained Math Math Logic
[Val92] Valmari A (1992) A stubborn attack on state explosion. Form Methods Syst Des 1(4):297–322
[YJ89] Yantchev J, Jesshope CR (1989) Adaptive, low latency, deadlock-free packet routing for networks of processors. IEE Proc E Comput Digit Tech 136(3):178–186
[YY91] Yeh WJ, Young M (1991) Compositional reachability analysis using process algebra. In: Proceedings of the symposium on testing, analysis, and verification. ACM, pp 49–59

Received 19 July 2018
Accepted in revised form 16 April 2019 by Eerke Albert Boiten
Published online 13 May 2019