Efficient verification of concurrent systems using synchronisation analysis and SAT/SMT solving

PEDRO ANTONINO, Department of Computer Science, University of Oxford, UK
THOMAS GIBSON-ROBINSON, Department of Computer Science, University of Oxford, UK
A. W. ROSCOE, Department of Computer Science, University of Oxford, UK

This paper investigates how the use of approximations can make the formal verification of concurrent system scalable. We propose the idea of *synchronisation analysis* to automatically capture global invariants and approximate reachability. We calculate invariants on how components participate on global system synchronisations and use a notion of consistency between these invariants to establish whether components can effectively communicate to reach some system state. Our synchronisation-analysis techniques try to show either that a system state is unreachable by demonstrating that components cannot agree on the order they participate in system rules, or that a system state is unreachable by demonstrating components cannot agree on the number of times they participate on system rules. These fully automatic techniques are applied to check deadlock and local-deadlock freedom in the *PairStatic* framework. It extends *Pair* (a recent framework where we use pure pairwise analysis of components and SAT checkers to check deadlock and local-deadlock freedom) with techniques to carry out synchronisation analysis. So, not only can it compute the same local invariants that *Pair* does, it can leverage *global* invariants found by synchronisation analysis, thereby improving the reachability approximation and tightening our verifications. We implement *PairStatic* in our DeadlOx tool using SAT/SMT and demonstrate the improvements they create in checking (local-)deadlock freedom.

CCS Concepts: • **Theory of computation** → **Concurrency**; **Invariants**; • **Software and its engineering** → **Model checking**; **Automated static analysis**;

Additional Key Words and Phrases: synchronisation analysis, approximate verification, reachability approximation, deadlock freedom, local-deadlock freedom, concurrent systems, SAT solving, SMT solving, invariants.

ACM Reference Format:
Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. 2018. Efficient verification of concurrent systems using synchronisation analysis and SAT/SMT solving. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (August 2018), 45 pages. https://doi.org/0000001.0000001

Authors' addresses: Pedro Antonino, Department of Computer Science, University of Oxford, Keble Road, Oxford, Oxfordshire, UK, pedro.antonino@cs.ox.ac.uk; Thomas Gibson-Robinson, Department of Computer Science, University of Oxford, Keble Road, Oxford, Oxfordshire, UK, thomas.gibson-robinson@cs.ox.ac.uk; A. W. Roscoe, Department of Computer Science, University of Oxford, Keble Road, Oxford, Oxfordshire, UK, bill.roscoe@cs.ox.ac.uk.

1049-331X/2018/8-ART $15.00
https://doi.org/0000001.0000001

1 INTRODUCTION

Many modern systems are built as a combination of components that cooperate to perform some intricate task. This need for cooperation is normally a great source of complexity in analysing these concurrent systems. Components can interact in many different ways, leading to many possible system behaviours, and we need to ensure they properly cooperate on each of these possible behaviours. For instance, proper cooperation often entails *deadlock freedom*: ensuring the system cannot reach a point in which all components are stuck. In this context, testing, a particularly useful technique in improving the quality of systems in the sequential world, is no longer as effective. It generally cannot account for most of these possibilities and, in a practical level, it is difficult to control the combination of interactions we want to test [25]. Formal verification is an alternative to testing that tends to be more effective in ensuring the quality of concurrent systems [31, 80]. While testing generally only ensures that the system behaves correctly in some cases, formal verification relies on mathematical foundations to ensure it always behaves appropriately.

Fully-automated verification techniques [14, 30, 40] automatically yield whether some input system description satisfies a given property or not. In the negative case, they also normally provide a counter-example, namely, a possible behaviour of the system that violates the given property. Counter-examples provide a very useful sort of information that helps refine the system's design, i.e. correct some bugs, so it can meet its intended specification. In this paper, we propose a fully-automatic verification framework for concurrent systems where components interact by exchanging messages (synchronising). These techniques have been widely and successfully integrated in the framework of model-driven development (MDD). This methodology's focus on the system modelling phase makes it a natural target for formal verification, which is used to ensure properties of system models [45, 77]. There are examples of the integration fully-automated verification frameworks for the analysis of UML and SysML diagrams [32, 36, 43, 54–56], component-based systems [15, 38, 62], and robotic systems [26, 59]; many of these frameworks are based around formalisms similar to ours, so they could directly benefit from the techniques presented in this work.

Verification tools, and specifically model checkers, normally rely on state-space exploration to prove system properties. For a concurrent system, the state space is given by the reachable combinations of component states, one per component. So, these tools construct and explore this product space to make sure the system cannot reach a *bad* state, i.e. a system state that violates the input property. In practice, these tools are heavily affected by the *state-space explosion problem* [14]: the state spaces of concurrent systems normally grow exponentially with the number of components. So, even the verification of relatively small systems can be infeasible. Intuitively, the vast number of ways in which components can cooperate leads to many different combinations of component states being reached. This issue has been a long-standing obstacle for tools verifying concurrent systems. The techniques that we introduce in this work are aimed at taming this problem.

Many techniques for tackling the state-space explosion problem have been invented; most notably partial-order reductions [47, 48, 67, 79], compression techniques (or, compositional reachability analysis) [75, 81], symbolic state-space representation [21, 24, 66] and counter-example-guided abstraction refinement (CEGAR) [27, 29]. These techniques employ different mechanisms to reduce the search space to be explored but they have in common their quest for a (sufficiently) precise reduction/abstraction of the original space. The exact nature of these reductions means they are bound to be even less efficient than simple explicit

state-space exploration in some cases, namely, when the reduction does not compensate the time taken to apply it. We have encountered many of these cases in practice.

Approximate techniques are alternatives to these exact approaches [12, 28, 41, 42, 57, 63, 64]. These 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, $\mathcal{P}'$ must be easier to check than $\mathcal{P}$. The first condition ensures soundness. If such a framework shows $\mathcal{P}'$, it can soundly deduce that $\mathcal{P}$ holds. Note, however, that the reverse implication ($\mathcal{P} \Rightarrow \mathcal{P}'$) need not hold; we allow approximate frameworks to be incomplete/imprecise. If $\mathcal{P}'$ does not hold for a system, we do not know whether $\mathcal{P}$ holds or not. The imprecision of these methods is deliberately introduced to gain efficiency. Instead of deciding the original, exact problem, one can look for an incomplete problem with a lower complexity. So, unlike exact methods, they should efficiently, and sometimes imprecisely, verify most input system.

A proxy property can be simply created by means of a *reachability over-approximation*. Many properties are naturally formulated as (or, can be simply translated into) "no bad state can be reached", where a bad state either exhibits or is a consequence of 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 is close to the actual state space of a system, it can give rise to a reasonably accurate approximate framework. Obviously, we expect the use of this approximation to speed up the verification process. In this work, we propose techniques to approximate reachability. We study them and, in particular, the deadlock-freedom and local-deadlock-freedom verification framework they give rise to. Local-deadlock freedom ensures that no subsystem of system under analysis becomes irretrievably blocked.

We propose the idea of *synchronisation analysis* to capture global invariants and approximate reachability. It relies on a data-flow-analysis-inspired framework, which we call component-synchronisation analysis (CSA), to calculate invariants on how components participate on (global) system synchronisations/interactions and on a notion of consistency between these invariants to establish whether components can effectively communicate to reach some system state. We introduce three synchronisation-analysis techniques: the first technique tries to show that a system state is unreachable by demonstrating that components cannot have agreed on the order they participate in system rules to get there, whereas the second and third techniques try to establish that a system state is unreachable by demonstrating components cannot agree on the number of times they have participated on system rules. These techniques are imprecise in the sense that they either establish that a system state is unreachable, or they are unable to do so and we conservatively assume the system state is reachable. This notion of cooperation consistency/feasibility for this combination captures *global invariants* of the system. Our CSA frameworks use abstract interpretation concepts [34, 35] that are normally part of data-flow-analysis frameworks [61] to compute invariants of LTSs that capture how components of a concurrent system participate on global synchronisations/interactions. This use is rather different to traditional frameworks that approximate values a variable might hold at different points of a program. As far as the authors of this paper are aware, our idea of synchronisation analysis and the techniques we implement are new.

In [2], we have investigated how *local analysis* can be leveraged to create reachability over-approximations, which are used by the framework *Pair* to check both local-deadlock and

deadlock freedom. Frameworks that are purely based on local analysis, however, are unable to prove properties that depend on some (global) invariant emerging from the global behaviour of the system. So, we combine our synchronisation-analysis techniques with *Pair* to create *PairStatic*. Thanks to the use of synchronisation analysis, *PairStatic* can capture some global invariants, improving on *Pair*'s precision. For instance, it can capture some global invariants of systems behaving like a systolic array or implementing a token mechanism. Since our framework tackles *NP*-hard problems, we build on the SAT encoding proposed for *Pair* to create efficiently checkable SAT and SMT encodings of our verification problems. We extend our DeadlOx tool [6] to implement this new framework. Furthermore, we demonstrate by a series of practical experiments that this tool is more accurate than (and as efficient as) similar approximate techniques. These experiments also suggest that the sort of reachability constraints derived by synchronisation analysis can be efficiently tackled by SAT/SMT solving. In this paper, we focus on deadlock and local-deadlock analysis but we have also investigate how synchronisation analysis can be applied to the verification of more general properties [1] and intend to cover these in sequels to this paper.

This paper extends our initial work on synchronisation analysis in [3] as follows.

- We more thoroughly describe and discuss our component-synchronisation-analysis framework. Moreover, we generalise our technique based on the relational consistency for the number of rule occurrences to the difference-based approach. This generalisation allowed us to also create the new notion of component-specific abstractions, which has proven very valuable to precisely analysing some systems. This generalisation was used to create a completely new characterisation and an associated SMT-based implementation for our *PairStatic* framework.
- We verify local-deadlock freedom, a property we did not handle in our original paper.
- We formally analyse the complexity of our reachability approximations and the *PairStatic* framework.
- We have formally proved our results, most of which were left unproven in the original work.
- We have evaluated our tools on further examples and against a few other techniques.

Outline. This paper's outline is as follows. Section 2 introduces the notion of supercombinator machines, upon which this work is based. We also briefly introduce the *Pair* framework for (local-)deadlock-freedom checking. In Section 3, we introduce the idea of synchronisation analysis. We introduce the concept of component-synchronisation analysis and propose and study three frameworks that implement synchronisation analysis, demonstrating how they can approximate reachability. Section 4 introduces *PairStatic*, a framework that combines *Pair*'s local analysis with the synchronisation-analysis techniques proposed in this paper. Finally, in Section 6, we present our concluding remarks.

2 BACKGROUND

In this section, we introduce *supercombinator machines*, the notation upon which our work is based. This notation is used by FDR4 [46] to implement CSP systems [50, 71]. Communicating Sequential Processes (CSP) [50, 71] is a notation used to model concurrent systems where processes interact by exchanging messages, and FDR4 is a refinement checker for CSP. 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 [71].

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 $(s, a, s') \in \Delta$, 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 rules to explicitly generate the LTS of a system [68], FDR4 relies on a *combinator-based* operational semantics [71] 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.

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

- $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ is a sequence of component LTSs;
- $\mathcal{R}$ is a set of rules of the form (e, a) where:
  - $e \in (\mathcal{E} \cup \{\tau, -\})^n$ specifies the event that each component must perform, where $-$ indicates that the component performs no event, and at most two process participate on a rule, namely, $\mathit{triple\_disjoint}(e)$ must hold, where $\mathit{triple\_disjoint}(e) = \forall\, i, j, k : \{1 \ldots n\} \mid i \neq j \wedge j \neq k \wedge i \neq k \bullet e_i = - \vee e_j = - \vee e_k = -$.
  - $a \in \mathcal{E} \cup \{\tau\}$ is the event the supercombinator machine performs.

FDR4 works with a version of a supercombinator machine that might have multiple *formats* and rules can have any number of participating components. 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 paper, however, we have the soft restriction that we only deal with single-format triple-disjoint machines. We call it a soft restriction because a multi-format non-triple-disjoint machine can be translated into a single-format triple-disjoint machine with an equivalent behaviour in polynomial time. Nevertheless, we impose this soft restriction because the techniques we propose should be better suited to handle systems that are naturally described by such restricted machines. In practice, many systems are naturally modelled by single-format triple-disjoint machines. Moreover, these restrictions are also imposed by many notations and frameworks that are similar to ours [7, 8, 12, 42, 58, 69]. 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. Milner's scheduler is a system where a token rotates amongst components and the component possessing the token is scheduled to work. In our model, event c_i represents the passing of a token from component $L_{i \ominus 1}$ to L_i (where $\ominus$ is subtraction modulo 2), a_i the work of component L_i, and b_i that component L_i is in an idle state. The system with 2 components is represented by the supercombinator machine $\mathcal{S}_{MS_2} = \{\langle L_0, L_1 \rangle, \mathcal{R}\}$, where $\mathcal{R} = \{((a_0, -), a_0), ((b_0, -), b_0), (-, a_1), a_1), ((-, b_1), b_1), ((c_0, c_0), c_0), ((c_1, c_1), c_1)\}$, and L_0 and L_1 as in Figure 1. ■

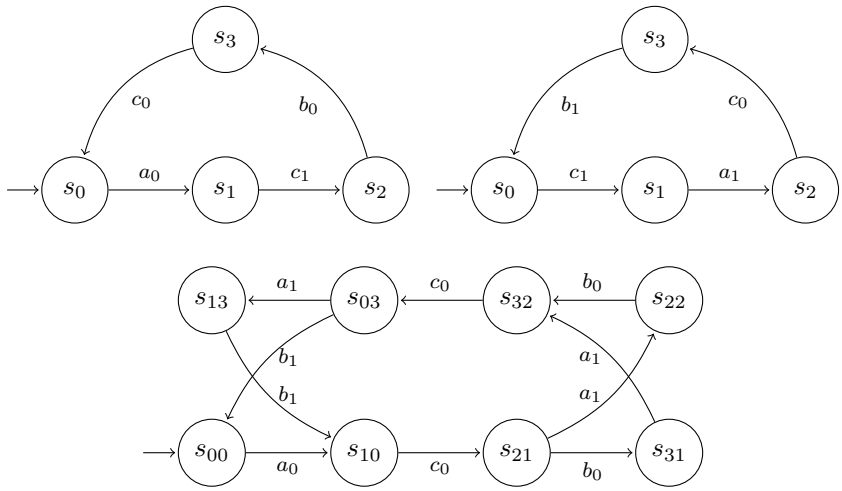

Fig. 1. LTSs for components L_0 and L_1 above that of $\mathcal{S}_{MS_2}$.

A supercombinator machine is an implicit representation of a system in the sense that it *induces* an LTS representing its behaviour. The LTS induced by $\mathcal{S}_{MS_2}$, for instance, is presented in Figure 1; we use s_{ij} to denote state (s_i, s_j).

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

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

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 representation interchangeably. So, we point out that 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 4. For induced LTS $(S, \Sigma, \Delta, \hat{s})$, state $s \in S$ is reachable if and only if $reachable(s)$ holds, where $reachable(s) = \exists\, p : \Sigma^* \bullet \hat{s} \xrightarrow{p} s$.

We have chosen supercombinator machines to reason about concurrent and distributed systems because they are simple and can seamlessly capture the behaviour of systems described in many common formalisms. We have used CSP to model our example systems and FDR4 to compile them into supercombinator machines but our frameworks 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. Furthermore, this operational notion, as intended, provides a system description that is fairly simple to implement and manipulate when constructing analysis tools.

In this work, we propose an approximate framework to verify *deadlock and local-deadlock freedom*. For the sake of decidability, we only consider supercombinator machines with a finite number of components, which are themselves represented by finite LTSs.

A system deadlock occurs when components becomes *blocked*, namely, they are unable to perform any further event. So, a system is deadlock free if no such state is reachable.

Definition 5. 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 $\neg\, \exists\, s : S \bullet \mathit{deadlock}(s)$ holds.

- $\mathit{deadlock}(s) = \mathit{reachable}(s) \wedge \mathit{blocked}(s)$
- $\mathit{blocked}(s) = \neg\, \exists\, e : \Sigma \bullet s \xrightarrow{e}$

A local deadlock, on the other hand, occurs when some subsystem becomes irretrievably blocked during the execution of the complete system. We determine whether a subsystem ss is irretrievably blocked by examining transitions induced by the projected set of 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 $r \restriction ss$ disregard (does not require) the participation of system components not in ss.

Definition 6. 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.

- $\mathit{local\text{-}deadlock}(s) = \mathit{reachable}(s) \wedge \mathit{locally\text{-}blocked}(s)$
- $\mathit{locally\text{-}blocked}(s) = \exists\, ss : \mathbb{P}(\{1 \ldots n\}) \mid ss \neq \emptyset \bullet \mathit{blocked}_{ss}(s)$
- $\mathit{blocked}_{ss}(s) = \neg\, s \xrightarrow{\mathcal{R}_{ss}}$, where $s \xrightarrow{\mathcal{R}_{ss}}$ is the predicate $s \longrightarrow$ 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 $r = ((e_1, \ldots, e_n), a)$.
  - $\mathit{on}(r, ss) = \exists\, i : ss \bullet e_i \neq -$
  - $r \restriction ss$ gives rise to tuple $((e'_1, \ldots, e'_n), a)$ where $e'_i = e_i$ if $i \in ss$ and $e'_i = -$ otherwise.

Deadlock freedom is often considered the first step towards showing that a concurrent system is correct. Moreover, many safety properties can be reduced to verifying deadlock freedom of modified systems [48]. In many cases, however, system designers are actually interested in achieving local-deadlock freedom. They want all components collaborating to the overall system behaviour, instead of having a single working component while others are forever stuck. So, verifying local-deadlock freedom seems like a more discerning way of checking for basic design flaws of concurrent systems if compared to verifying deadlock freedom.

We point out that local-deadlock freedom 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, we point out that local-deadlock freedom implies deadlock freedom as the entire system is indeed one of the analysed subsystem. Of course, the converse does always not hold.

In [6], we have shown that the problems of checking both deadlock- and local-deadlock-freedom are *PSPACE*-complete. Intuitively, the need to explore induced LTSs and their state-space explosion can be seen as the cause for this problem's membership of this complexity class. 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, current algorithms employed to solve *PSPACE*-complete problems take exponential time. Thus, automatic verification techniques should struggle to show deadlock and local-deadlock freedom even for systems with a reasonably small number of components. FDR4, for example, is a fully-automatic verification tool that checks properties about CSP systems. It 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. It does not, however, provide an assertion to check local-deadlock freedom.

2.1 A brief introduction to the Pair framework

In [2], we proposed *Pair*; a framework that relies on a local-analysis-based reachability approximation to check deadlock and local-deadlock freedom. It analyses how pairs of components interact using the following projection. Intuitively, it analyses how a pair of components cooperate if they are isolated from the rest of the system.

Definition 7. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. The *pairwise projection* $\mathcal{S}_{i,j}$ of the machine $\mathcal{S}$ on components i and j is given by:

$$\mathcal{S}_{i,j} = (\langle L_i, L_j \rangle, \{((e_i, e_j), a) \mid ((e_1, \ldots, e_n), a) \in \mathcal{R} \bullet (e_i \neq - \vee e_j \neq -)\})$$

Pair characterises a deadlock (local deadlock) as a state of the system that is blocked (locally blocked) and fully consistent with pairwise reachability information. We called it a *Pair candidate* (*Pair local candidate*).

Definition 8. 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 $reach_2(s)$ and $blocked(s)$ hold;
- a *Pair local candidate* if and only if $reach_2(s)$ and $locally\_blocked(s)$ hold.
  - $reach_2(s) \mathrel{\widehat{=}} \forall i, j \in \{1 \ldots n\} \mid i \neq j \bullet reachable_{i,j}((s_i, s_j))$.

$reachable_{i,j}$ is the reachable predicate for the pairwise projection $\mathcal{S}_{i,j}$.

Our pairwise-reachability predicate ($reach_2$) captures the intuitive notion that if a pair of components cannot effectively cooperate to reach a system state (or, rather their respective component states) in isolation, they cannot do so when they are considered in the context of the system. We also refer to this notion as 2-reachability. This approximation can successfully show deadlock and local-deadlock freedom for some well-behaved resource-allocation and client-server systems. The use of pure local (pairwise) analysis, however, means that this predicate is unable to show unreachability (and consequently deadlock and local-deadlock freedom) if that is due to some invariant emerging from the system's global behaviour. The work in this paper is a means to address (some of) this inability. We propose the combination of some global invariants derived using synchronisation analysis with these local invariants calculated by Pair, thereby tightening the approximate state space that needs to be analysed in the verification task at hand.

3 APPROXIMATE REACHABILITY VIA SYNCHRONISATION ANALYSIS

In this section, we present the key ingredients behind a framework for *synchronisation analysis*. We begin by proposing the concept of a component-synchronisation-analysis framework, demonstrating how it can be used to calculate invariants of components that capture how they participate on global interactions, and hinting at how they can be combined to test reachability. Then, we demonstrate how these ingredients can be combined to create

Fig. 2. Example of components L_1 and L_2.

three synchronisation-analysis techniques that can effectively capture some global system invariants. We study the consistency notions (on component invariants) that they use to approximate reachability and the sorts of invariants and concurrency mechanisms these consistency notions can capture.

3.1 Component-synchronisation analysis

Our techniques rely on a *component-synchronisation-analysis* (CSA) framework to calculate *component-state invariants*. A component-state invariant compactly and conservatively summarises the behaviour leading a component to one of its states. It summarises the participation of this component in (global) interactions/synchronisations. Our CSA framework uses elements from abstract interpretation [34, 35] in a way similar to what data-flow-analysis (DFA) frameworks do; we were particularly inspired by [61]. DFA frameworks use abstract interpretation to capture the values of program variables at different program points by analysing the program's control-flow graph. Here, we use abstract interpretation to compute some piece of information about a component's participation on system synchronisations at different component states by analysing its LTS. More concretely and formally, we compute an over-approximation for the values a function $f(tr)$ might give when applied to the component's traces; the function captures the piece of information we want to analyse. This over-approximation is captured by an *informative variable*, i.e. a "ghost" variable that is not actually part of our formalism/component definition. Note that our formalism does not have the notion of a program/component variable. To illustrate these concepts, throughout this section, we create a CSA framework to estimate the numbers of times component L_1 (or L_2) in Figure 2 performs event a to reach each of its states; we use the (informative) variable N_a to over-approximate the values that function $f_a(tr) = tr \downarrow a$, where $tr \downarrow e$ counts the number of events e in trace tr, might take at its different states.

A CSA is defined by a triple $(D, T_e, Init)$ where D is an *abstract domain*, T_e is an *abstract transformer* monotone on D, and $Init$ is an *initial value* in D. The abstract domain $D = (S, \sqsubseteq)$ is a complete lattice with finite height where S is the set of possible values for the variable (over-approximation) under analysis, and $\sqsubseteq$ is an order on S such that the greater the value the more information it carries. The transformer T_e is a function that calculates how the information we are computing changes when the event e is performed. That is, $T_e(v)$ gives the variable's value after the event e is performed from a component state where the variable's value is $v \in D$. $Init \in D$ is a value depicting the initial value of the variable, before any event is performed by the LTS. We use the flat integer domain to represent the values of variable (over-approximation) N_a. This domain is given by $D = (S, \sqsubseteq)$ where S is a set containing all singleton sets of integers, the set of all integers (i.e. $\top = \mathbb{Z}$) and the empty set (i.e. $\bot = \emptyset$), and $\sqsubseteq$ the usual subset order on sets. The transformers could be given by $T_e(\mathbb{Z}) = \mathbb{Z}$, $T_e(\emptyset) = \emptyset$, and $T_a(\{v\}) = \{v+1\}$ and $T_e(\{v\}) = \{v\}$ for $e \neq a$. Finally, we could define $Init = \{0\}$, that is, initially at the starting state the component has performed no as.

Given a CSA triple and a LTS, this framework derives a set of equations that define a fixed-point X, that is, a collection of domain values satisfying the following equations, where $\sqcup$ denotes the join operator induced by D and $\hat{s}$ the LTS's initial state.

- $X_{\hat{s}} = Init \sqcup X_{\hat{s}}$;
- $X_{s_j} = T_e(X_{s_i}) \sqcup X_{s_j}$, for each transition (s_i, e, s_j) in the LTS.

X is a collection that has an element X_{s_i} per state s_i in the LTS. X_{s_i} gives the value of the variable under analysis considering state s_i, namely, it over-approximates the value that f might take in the sense that (i) for any trace tr leading the component to s_i it must be the case that $f(tr) \sqsubseteq X_{s_i}$ holds; (i) is the component-state invariant computed by CSA.

Each (right-hand side) of these equations capture the effect of a transition on the values of variables; the value of variables in the target state is derived from the value of variables in the source state and the transition's event. Considering the CSA-example triple we proposed and component L_1, we have, for instance, equation $X_{s_0} = \{0\} \cup X_{s_0}$ to account for the starting transition, which captures that the number of as performed must start at zero; equation $X_{s_1} = T_b(X_{s_0}) \cup X_{s_1}$ captures transition (s_0, b, s_1), namely, it captures that N_a's value in s_1 might be derived from its value in s_0 and the performing of event b; and so on.

A least fixed-point can be calculated for these equations using the standard worklist algorithm [61]. It starts with X as the collection of bottom elements of the abstract domain and it iteratively uses (fairly) the right-hand side of these equations as "recipes" to update the corresponding left-hand side elements. Considering the CSA triple we proposed and component L_1, if initially $X_{s_0} = \emptyset$, we can use equation $X_{s_0} = \{0\} \cup X_{s_0}$ to update X_{s_0} to $\{0\}$; if $X_{s_0} = \{0\}$ and $X_{s_1} = \emptyset$, we can use equation $X_{s_1} = T_b(X_{s_0}) \cup X_{s_1}$ to update X_{s_1} to $\{0\}$; and so on. The least fixed-point gives the most precise approximation (amongst fixed-points) for variable values. This iterative process eventually reaches the least fixed-point thanks to the Knaster-Tarksi theorem [78] and to the finite height of the domain. Using our CSA-example framework, we have for component L_1 the (least) fixed-point collection $X_{s_0} = \{0\}$, $X_{s_1} = \{0\}$, $X_{s_2} = \{1\}$; and for component L_2 the (least) fixed-point collection $X_{s_0} = \{0\}$, $X_{s_1} = \{1\}$, $X_{s_2} = \{2\}$.

Note that fixed-point collection X can be calculated in polynomial time on the number of nodes $|S|$ of the LTS, the number of the transitions $|\Delta|$, and the height h of the lattice in the CSA triple, provided that transformer and join operations take time $\mathcal{O}(h)$.

Theorem 1. *The worklist algorithm can calculate a fixed-point X in $\mathcal{O}(|V| \cdot |E| \cdot h^2)$.*

Proof. We over-estimate the number of steps the worklist algorithm can take to reach a fixed-point as follows. In our CSA framework, we have an equation per transition so we have $|\Delta|$-many equations. We call an *iteration* of the worklist algorithm, a round of updates where it goes over each equation and carries out the update of the element indicated by its left-hand side using the "recipe" on the right-hand side. If a fixed-point has not been reached, (i) at least an element must be modified. Each iteration, then, takes $\mathcal{O}(|\Delta| \cdot h)$ steps, as each equation leads to an update taking time $\mathcal{O}(h)$ due to the use of join and transformer. Moreover, each element X_{s_i} in this collection can be modified at most h times, since each modification has to assign X_{s_i} to a higher value in the abstract domain lattice. So, (ii) there are $\mathcal{O}(|S| \cdot h)$ modifications possible. Putting together, (i) and (ii), the worklist algorithm can iterate at most $\mathcal{O}(|S| \cdot h)$ times each of which takes $\mathcal{O}(|\Delta| \cdot h)$ steps. □

Assuming for instance that components L_1 and L_2 are placed in a system where they run in parallel and need to synchronise on shared events, we can use the component-state invariants computed using our example CSA framework to test (un)reachability. For instance,

state (s_2, s_2) cannot be reached because while L_1 needs to perform a exactly once to get to s_2, L_2 needs to perform a exactly twice to get to s_2. As they need to synchronise on shared events, they can only reach combination of component states where they agree on the number of performed as. This sort of (in)consistency checking is what lies behind the reachability testing implemented by the synchronisation analysis we now introduce.

3.2 Synchronisation-analysis techniques

Our synchronisation-analysis techniques combine component-state invariants to approximate reachability. Informally, our techniques try to show that, based on their individual behaviour, components cannot cooperate to reach a given system state and so the state must be unreachable. More precisely, the (global) interactions components must engage on to reach the system state under analysis, captured by their component-state invariants, are analysed in an attempt to show that components are unable to consistently participate in system rules that would lead the system to this state. The first technique tries to show that components cannot agree on the order they participate in system rules, while the second and third techniques try to establish that components cannot agree on the number of times they participate in system rules.

Our techniques either establish a system state is unreachable, or they are unable to do so and we conservatively, and maybe imprecisely, assume it is reachable. This imprecision is mainly due to the approximative nature of component-state invariants. They are meant to be a compact and sound approximation for a set of behaviours of a component and this comes at the price of imprecision. We point out also that the sort of consistency analysis these techniques perform over all components of the system can capture some (global) invariants emerging from the global behaviour of the system. So, they can prove unreachability for some system states that are beyond the capabilities of techniques relying on pure local analysis.

These techniques analyse the behaviour components though a *rule-participation projection*. This projection depicts a component's behaviour in terms of the system rules in which it can participate rather than its own events. To capture this projection, we assume that system rules are identified by some k, that is, $\mathcal{R} = \{r_1, \ldots, r_m\}$ so r_k gives some system rule. We also reuse r_k as fresh events to annotate transitions that involving rule r_k. The use of r_k both as a rule and an event should cause no confusion.

Definition 9. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine. We use $r_k = (e, a)$ to denote that rule (e, a) is identified by k. The rule-participation projection of $\mathcal{S}$ over i is given by $\mathcal{S}_i = (\langle L_i\rangle, \{((e_i), r_k) \mid r_k = (e, a) \in \mathcal{R} \bullet e_i \neq -\})$.

So, the LTS induced by $\mathcal{S}_i$ replaces transitions with event e in the original component LTS by transitions annotated with the rules component i can participate in using event e. So, a trace for this LTS is a sequence of rules that component i might engage on, whereas its alphabet gives the set of rules (or, rule events) this component can participate in. In a straightforward way, these projections can be used to assess reachability for the original system; if component projections can cooperate on shared rule events to reach a state, it must be the case that the same cooperation can be achieved by components of the original system leading to the same state. From our definition, it should be clear that a rule-participation projection and its induced LTSs can be computed in polynomial time on the size of the input supercombinator machine.

This explicit cooperation through *unified* rule events is paramount for the techniques presented in this paper. Rule events provide a common frame of reference to compare the

behaviour of components, and consequently component-state invariants. Supercombinator machines generally allow rules to involve a different event per component or a single component event to participate in multiple rules. This generality means that using the same two events, two components might be able to engage in multiple rules, and this multitude of possibilities increases the complexity of the sort of component-behaviour consistency we check in our synchronisation-analysis frameworks. For instance, if we have two traces tr_i and tr_j of components i and j, respectively, and we want to know if these traces can be combined to create a sequence of valid rule applications, in general, we would need to try out multiple system rules to test that. With our unified events, however, it suffices to check whether $tr_i \upharpoonright \Sigma'_j = tr_j \upharpoonright \Sigma'_i$, where $tr_i \upharpoonright \Sigma'_j$ gives the sequence resulting from removing from tr_i elements that are not in Σ'_j, and Σ'_i and Σ'_j are the alphabets of the LTSs induced by projections $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively.

3.2.1 Ordering of rule occurrences consistency. The first technique we propose tries to show that a system state is unreachable by showing that components cannot agree on the *order* in which they cooperate to reach this state. We use Example 2 as a running example to explain this technique.

Example 2 (From [71])**.** Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 3 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. For the sake of presentation, we use the name of an event to refer to the rule that requires its synchronisation. For instance, $r_{ring_1} = ((ring_1, ring_1, -), ring_1)$. As τ is not synchronised, there are three rules τ_0, τ_1, τ_2, such that τ_i allows component i to perform a τ. This machine describes a ring-like message-exchange system. A component can receive messages either from its predecessor component in the ring via event $ring_i$, or from its user via event in_i. If it holds a message, it can pass the message along to the next component in the ring, via event $ring_{i\oplus 1}$, or output the message to its user, via out_i. The τ transitions represent an internal (non-deterministic) decision of the component.

The *blocked* (and consequently *locally blocked*) system state (s_6, s_6, s_6) is unreachable as components cannot cooperate to reach it. Each component in this system can hold up to two messages, and the message that makes a component full can only come from its predecessor in the ring. So, when full, the most recent action of a component must have been the receiving of a message from its predecessor in the ring. We can show that state (s_6, s_6, s_6) is unreachable by using this fact about full component states. Since each component is full,

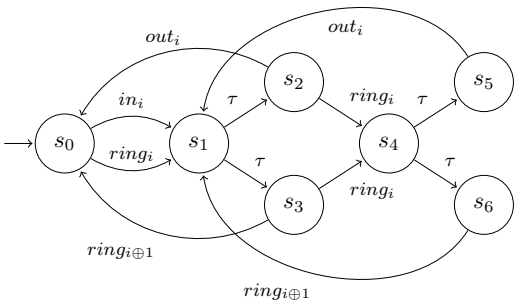

Fig. 3. LTS of component L_i where $\oplus$ represents addition modulo 3.

we know that component i's last action must have happened after component $i \oplus 1$'s last action. So, by going around the ring, we can derive the contradiction that component i's last action must have happened after component $i \oplus 1$'s last action. This contradiction shows that components cannot effectively interact to reach this state.

This reasoning sketches the sort of analysis that we try to capture with the technique proposed in this section. The fact about full states of components is a component-state invariant. So, a combination of component-state invariants, one per component state in this case, is used to check whether components can effectively interact to reach a system state. Throughout this section, we detail how our technique systematically deduces this contradiction. We point out that our techniques can capture this reasoning for similar non-fillable rings with more than three components. Also, note that this system state is unreachable and yet it is 2-reachable. The invariant "all components cannot be simultaneously full" captured by our combination of component-state invariants emerges from the system's global behaviour and, thus, it cannot be derived by pure local analysis. ■

To show that components cannot agree on such an order, this technique relies on a sequence $SF_{i,s}$ of rule events as the component-state invariant for state s of component i. This sequence is the longest common suffix for all traces tr leading the component i's projection to s. We employ the following CSA framework to systematically calculate $SF_{i,s}$. Note the partial behaviour given by this suffix summarises all (possibly infinitely many) sequences of rule applications that can lead component i to its state s.

Definition 10. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$. When applied to L'_i, the following static analysis framework computes a collection SF_i of $|S'_i|$ sequences of rule events, where $SF_{i,s} \in ((\Sigma'_i)^* \cup \{\bot\})$ is a common suffix for all traces of L'_i leading to $s \in S'_i$.

- $Init = \langle\rangle$
- $D = (\{\bot\} \cup \Sigma_i'^{|S'_i|}, \sqsubseteq)$, where $a \sqsubseteq b$ holds if b is a suffix of a and $\bot$ is the least element, and $\Sigma_i'^{|S'_i|}$ is the set of sequences of events in Σ'_i with at most $|S'_i|$ elements.
- $T_{r_k}(\bot) = \bot$ and $T_{r_k}(v) = v\hat{}\langle r_k \rangle$.

Given these three elements and $\sqcup$, the join operator induced by the lattice D, the collection SF_i is the least fixed point for the following set of equations:

- $SF_{i,\hat{s}'_i} = Init \sqcup SF_{i,\hat{s}'_i}$
- $SF_{i,s'} = T_{r_k}(SF_{i,s}) \sqcup SF_{i,s'}$, for each $(s, r_k, s') \in \Delta'_i$

We capture how a component participates in the overall system behaviour using an *occurrence suffix* that we derive from a component-state invariant as follows. Such suffixes use occurrence variables o_k^l to denote the l-th most recent system-wide occurrence of rule k, namely, o_k^l marks the point/moment in which all components should synchronise to perform the l-th most recent application of rule k.

Definition 11. $SO_{i,s} \mathrel{\widehat{=}} Occur(SF_{i,s})$ is the occurrence suffix derived from component-state invariant $SF_{i,s}$, where $Occur(\bot) \mathrel{\widehat{=}} \bot$ and $Occur(tr)$, for $tr \in (\Sigma'_i)^*$, gives the sequence of occurrence variables that is obtained by replacing the l-th most recent occurrence of rule r_k in tr by o_k^l. We use $SO_{i,s,j}$ to denote the occurrence variable in position j in $SO_{i,s}$ where $j \in \{1 \ldots m_{i,s}\}$ and $m_{i,s}$ gives the size of $SO_{i,s}$.

For instance, state (s_6, s_6, s_6) of the system in Example 2 gives rise to the following component-state invariants and occurrence suffixes: $SF_{0,s_6} = \langle \tau_0, ring_0, \tau_0 \rangle$, $SF_{1,s_6} =$

$\langle \tau_1, ring_1, \tau_1 \rangle$, $SF_{2,s_6} = \langle \tau_2, ring_2, \tau_2 \rangle$, $SO_{0,s_6} = \langle o^1_{\tau_0}, o^0_{ring_0}, o^0_{\tau_0} \rangle$, $SO_{1,s_6} = \langle o^1_{\tau_1}, o^0_{ring_1}, o^0_{\tau_1} \rangle$, and $SO_{2,s_6} = \langle o^1_{\tau_2}, o^0_{ring_2}, o^0_{\tau_2} \rangle$.

Note that these suffixes can be used to compare the order in which components participate in system-wide rule occurrences. So, we can use them to establish whether components can agree on an order in which they participate in these occurrences.

For system state $s = (s_1, \ldots, s_n)$, this technique checks whether components can agree on a consistent order in which they can perform the rule occurrences in suffixes SO_{i,s_i} for $i \in \{1 \ldots n\}$, namely, it tests if there exists a linear ordering for these rule occurrences that respects their relative ordering in all these occurrence suffixes. This analysis is captured by predicate $reach_S$, which uses the clock variables clk^l_k, marking the instant at which the occurrence o^l_k happened, to find such a global ordering.

Definition 12. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$. To define our predicate we need the following auxiliary definitions: $O_{i,s} \mathrel{\widehat{=}} \{SO_{i,s,j} \mid j \in \{1 \ldots m_{i,s}\}\}$ gives the set of elements in $SO_{i,s}$, $O \mathrel{\widehat{=}} \bigcup_{i \in \{1 \ldots n\}, s \in S'_i} O_{i,s}$ the universal set of occurrences, and $O_i \mathrel{\widehat{=}} \{o^l_k \mid o^l_k \in O \wedge r_k \in \Sigma'_i\}$ the set of rule occurrences requiring the participation of component i. Finally, we use $clk_{i,s,j}$ to denote clk^l_k if $SO_{i,s,j} = o^l_k$.

Let $s = (s_1, \ldots, s_n)$ be a system state and $O = \{o^{l_1}_{k_1}, \ldots, o^{l_z}_{k_z}\}$ the universal set of occurrences. We propose the following predicate to approximate reachability.

$$reach_S(s) \mathrel{\widehat{=}} \exists\, clk^{l_1}_{k_1}, \ldots, clk^{l_z}_{k_z} : \mathbb{N} \bullet \bigwedge_{i \in \{1 \ldots n\}} HBC(i, s_i)$$

The constraint $HBC(i, s)$ creates a happens-before relation based on the suffix $SO_{i,s}$.

$$HBC(i,s) = \begin{cases} \mathit{false} & \text{if } SO_{i,s} = \bot \\ \mathit{true} & \text{if } SO_{i,s} = \langle\rangle \\ TC(i,s) \wedge BC(i,s) & \text{otherwise} \end{cases}$$

If $SO_{i,s} = \bot$, then state s is unreachable in L'_i. So, this state cannot be part of a reachable system state, and we create the unsatisfiable constraint *false*. Non-empty suffixes contribute to create our happens-before relation, while empty ones do not. Hence, the creation of the always-satisfied constraint *true* for the latter. A non-empty occurrence suffix gives rise to the conjunction of the *trace constraint* $TC(i, s)$ and the *before constraint* $BC(i, s)$. $TC(i, s)$ enforces that a valid system behaviour respects the order in which rule occurrences appear in $SO_{i,s}$:

$$TC(i,s) \mathrel{\widehat{=}} \bigwedge_{j \in \{1 \ldots m_{i,s}-1\}} clk_{i,s,j} < clk_{i,s,j+1}$$

As for $BC(i, s)$, it captures that all occurrences of rules requiring the participation of component i but not in $SO_{i,s}$ must have happened before the occurrences in $SO_{i,s}$:

$$BC(i,s) \mathrel{\widehat{=}} \bigwedge_{o^l_k \in O_i - O_{i,s}} clk^l_k < clk_{i,s,1}$$

If this predicate holds, there exists a consistent sequence of rule occurrences that represents a valid (likely partial) system behaviour on which components can agree. There is no guarantee that this sequence is a total system behaviour as the rule occurrences being analysed might not lead the system from its initial state all the way to the system state being tested. Therefore, we can neither guarantee that the state is reachable nor unreachable, and so, we conservatively assume the former.

On the other hand, if the predicate does not hold, the HBC constraints are inconsistent either because a component state is trivially unreachable considering this component's projection, or because there is an inconsistency between components' happens-before orderings. The former trivially implies that the system state is unreachable, whereas the latter implies that components are unable to cooperate to perform the rule occurrences in the suffixes being analysed, and consequently, they cannot cooperate to reach this state.

For instance, state (s_6, s_6, s_6) of the system in Example 2 gives rise to the following happens-before constraints:

(1) $HBC(0, s_6) = clk^1_{\tau_0} < clk^0_{ring_0} \wedge clk^0_{ring_0} < clk^0_{\tau_0} \wedge clk^0_{ring_1} < clk^1_{\tau_0}$;
(2) $HBC(1, s_6) = clk^1_{\tau_1} < clk^0_{ring_1} \wedge clk^0_{ring_1} < clk^0_{\tau_1} \wedge clk^0_{ring_2} < clk^1_{\tau_1}$;
(3) $HBC(2, s_6) = clk^1_{\tau_2} < clk^0_{ring_2} \wedge clk^0_{ring_2} < clk^0_{\tau_2} \wedge clk^0_{ring_0} < clk^1_{\tau_2}$.

From 1, 2 and 3, we can deduce that $clk^0_{ring_0} < clk^0_{ring_2} < clk^0_{ring_1} < clk^0_{ring_0}$. This contradiction shows that $reachable((s_6, s_6, s_6))$ is false and that components cannot agree on a consistent ordering in which they participate on rule occurrences.

Since our component-invariant soundly summarises the behaviour of a component and components must synchronise on shared rules, it follows that, to reach a state, components must be able to, in particular, consistently synchronise on the rule occurrences in the occurrence suffixes derived.

Theorem 2. *For a system state s, $reachable(s) \Rightarrow reach_S(s)$.*

Proof. We assume that $s = (s_1, \ldots, s_n)$ is a reachable system state and show that there exists a collection of clock values that respects constraints $HBC(i, s_i)$. Let L'_i be the rule-participation projection of $\mathcal{S}$ on component i.

If s is reachable, we can assume without lost of generality that $tr = \langle r_{k_1}, \ldots, r_{k_w} \rangle$ is a sequence of rule applications (rule events) leading the system to s. From tr, we can calculate a corresponding occurrence sequence $Occur(tr) = \langle o^{l_1}_{k_1}, \ldots, o^{l_w}_{k_w} \rangle$. From this occurrence sequence, we can derive the assignment to clock variables: $clk^{l_1}_{k_1} = 1, \ldots, clk^{l_w}_{k_w} = w$. This assignment gives an ordering in which the system can perform these rule occurrences to reach s.

As tr is a valid sequence of rule applications for the system, the projection L'_i must be able to engage on $tr \restriction \Sigma'_i$ to reach state s_i, where $tr \restriction \Sigma'_i$ is the result of filtering out all rule occurrences in tr that are not in Σ'_i. Also, for each component i, the rule events in $tr \restriction \Sigma'_i$ respect their ordering in tr given how operator $\restriction$ works.

From Lemma 1, we know that SF_{i,s_i} is a suffix of $tr \restriction \Sigma'_i$. Putting these two facts together, we can see that rule events in SF_{i,s_i} respect their ordering in tr. So, the occurrence suffixes SO_{i,s_i} must respect the ordering of variables in $Occur(tr)$, and consequently, the clock constraints in $HBC(i, s_i)$ are satisfied by our assignment to clock variables. □

Lemma 1. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$, and SF_i computed as per Definition 10. $SF_{i,s}$ is a common suffix for the traces leading L'_i to its state s.*

Proof. We show that if a fixed-point SF_i is reached there cannot be a trace tr of rule events that leads L'_i to s such that $SF_{i,s}$ is not a suffix of tr. We demonstrate that the existence of such a trace tr would lead to a contradiction. We assume without lost of generality that tr is the shortest such trace.

Thanks to equation $SF_{i,\hat{s}'_i} = Init \sqcup SF_{i,\hat{s}'_i}$ and our fixed-point assumption, we know that $SF_{i,\hat{s}'_i} = \langle\rangle$. Clearly, then, tr cannot be $\langle\rangle$ as $tr = \langle\rangle$ only leads to $\hat{s}'_i$ and $SF_{i,\hat{s}}$ is indeed a suffix of $\langle\rangle$.

Thus, we can safely assume that tr is of the form $tr'^\frown\langle r_k\rangle$, where $\hat{s}_i \xrightarrow{tr'} s' \xrightarrow{r_k} s$. Moreover, thanks to our assumption that tr is the shortest trace, we know $SF_{i,s'}$ is a suffix for tr'. If $SF_{i,s'}$ is a suffix for tr', then the equation $SF_{i,s} = T_{r_k}(SF_{i,s'}) \sqcup SF_{i,s}$, corresponding to transition $s' \xrightarrow{r_k} s$, does not hold, contradicting our fixed-point assumption. While $T_{r_k}(SF_{i,s'}) \sqcup SF_{i,s}$ is a suffix of tr, $SF_{i,s}$ is not. □

We finish this section, by showing that $reach_S(s)$ can be checked in polynomial time on the size of the input supercombinator machine.

Lemma 2. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine such that $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $|L_{Max}|$ the size of the largest component in $\mathcal{S}$. For a given state system s of $\mathcal{S}$, we can decide $reach_S(s)$ in time $\mathcal{O}(n^2 \cdot |L_{Max}|^4 \cdot |\mathcal{R}|^2)$.*

Proof. Let $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ be the LTS induced by rule-participation projection $\mathcal{S}_i$, and SF_i computed as per Definition 10. Note that $|L'_i| = \mathcal{O}(|L_i| \cdot |\mathcal{R}|)$. The domain that we use in the CSA framework to calculate SF_i has height $h = \mathcal{O}(|L'_i|) = \mathcal{O}(|L_i| \cdot |\mathcal{R}|)$. So, from Theorem 1, we deduce it takes time $\mathcal{O}(|L_i|^2 \cdot (|L_i| \cdot |\mathcal{R}|)^2)$ to calculate SF_i, and time $\mathcal{O}(\sum_{i\in\{1\ldots n\}}(|L_i|^4 \cdot |\mathcal{R}|^2))$, which we approximate to $\mathcal{O}(n \cdot |L_{Max}|^4 \cdot |\mathcal{R}|^2))$, to calculate all fixed-points SF_i for $i \in \{1 \ldots n\}$.

To decide $reach_S(s)$, we check whether the digraph where nodes are clock variables and edges are induced based on the $<$-relation defined by constraints TC and BC is cycle free. We can roughly over-estimate the number of clock variables, and consequently the size of this digraph, based on the number of occurrence variables. The suffix for a state of component i can have at most $|L_i|$ elements, giving rise to $|L_i|$ occurrence variables. If all occurrences for all suffixes of component states in the system state under analysis are different, we have an upper-bound on occurrence (and clock) variables of $\mathcal{O}(\sum_{i\in\{1\ldots n\}} |L_i|)$, which we approximate to $\mathcal{O}(n \cdot |L_{Max}|)$. So, we roughly over-approximate the size of this digraph by $\mathcal{O}(n^2 \cdot |L_{Max}|^2)$. A digraph can be checked cycle free in linear time using a modified depth-first-search algorithm.

Thus, we have the loose upper-bound of $\mathcal{O}(n^2 \cdot |L_{Max}|^4 \cdot |\mathcal{R}|^2)$ on the time taken to decide $reach_S(s)$ for a fixed system state s. □

3.2.2 Number of rules occurrences – relational consistency. In the second technique, we try to show that a system state is unreachable by showing that components cannot agree on the *number of times* they need to cooperate to reach this state. We use Example 3 as a running example in introducing this technique.

Example 3. Let $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 4 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. For the sake of presentation, we use the name of an event to identify the rule requiring its synchronisation. This system implements a token ring where process L_0 passes the token initially to L_1 and the events tk_i represent the passage of a token from $L_{i\ominus 1}$ to L_i, where $\ominus$ is subtraction modulo 3.

The system state (s_3, s_2, s_2) is unreachable as components cannot cooperate to reach it. The system has a single token that gets passed around. Component 0 holds a token in states where it has performed events tk_0 and tk_1 the same number of times, whereas Component 1 (2) holds a token in states where it has performed more events tk_1 (tk_2) than tk_2 (tk_0,

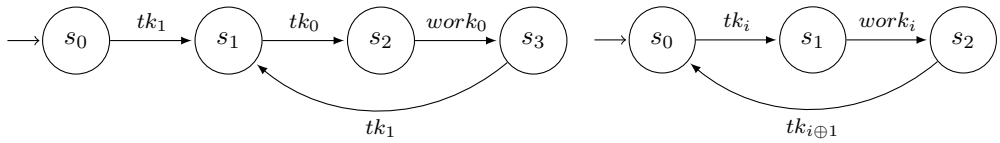

Fig. 4. LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

respectively). We can show that state (s_3, s_2, s_2) is unreachable by using these facts about states where components hold a token. In (s_3, s_2, s_2), all components hold a token. So, from Component 0's behaviour we can deduce that events tk_0 and tk_1 must have been performed the same number of times, whereas from the behaviour of Components 1 and 2, we can deduce that event tk_0 must have happened more times than tk_1. This contradiction shows that these components cannot effectively interact to reach this state.

This reasoning sketches the sort of analysis that we try to capture with our second and third techniques. The differences between the number of times events are performed are component-state invariants, and we use these invariants to check whether components can effectively interact to reach a system state. Throughout this section, we detail how our technique systematically carries out this sort of analysis. We point out that our techniques can capture this reasoning for similar token rings with more than three components. Finally, we point out that this system state is unreachable and yet it is 2-reachable. This technique captures, to some extent, that tokens are conserved and, thus, this state must be unreachable. The conservation of tokens, however, is a global system invariant, and as such, it cannot be captured by pure local analysis. ■

This technique relies on the set $DS^{k,l}_{i,s}$ of integers as a component-component state invariant for state s of component i. For the pair of rule events k, l in component i's projection, the difference between the number of rule events r_k and r_l in any trace tr leading component i's projection to its state s lies in $DS^{k,l}_{i,s}$. As follows, we propose a CSA framework that systematically calculates these *difference sets*.

Definition 13. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$. We propose a CSA framework that is parametrised by rules k and l, where $k \neq l$, that when applied to L'_i computes the collection $DS^{k,l}_i$ of sets with a set $DS^{k,l}_{i,s} \in (\{\emptyset, \mathbb{Z}\} \cup \{\{a\} \mid a \in \mathbb{Z}\})$ for each $s \in S_i$.

- $Init = \{0\}$;
- $D = (\{\emptyset, \mathbb{Z}\} \cup \{\{a\} \mid a \in \mathbb{Z}\}, \subseteq)$ the flat integer domain where $\subseteq$ is the usual order on sets;
- $T_{r_h}(\{d\}) \mathrel{\widehat{=}} \begin{cases} \{d+1\} & \textbf{if } h = k \\ \{d-1\} & \textbf{if } h = l \\ \{d\} & \textit{otherwise} \end{cases}$, $T_{r_h}(\emptyset) \mathrel{\widehat{=}} \emptyset$ and $T_{r_h}(\mathbb{Z}) \mathrel{\widehat{=}} \mathbb{Z}$.

Given these three elements and $\sqcup$, the join operator induced by the lattice D, the collection $DS^{k,l}_i$ is the least fixed point for the following set of equations:

- $DS^{k,l}_{i,\hat{s}'_i} = Init \sqcup DS^{k,l}_{i,\hat{s}'_i}$
- $DS^{k,l}_{i,s'} = T_{r_h}(DS^{k,l}_{i,s}) \sqcup DS^{k,l}_{i,s'}$, for each $(s, r_h, s') \in \Delta'_i$

For instance, for state (s_3, s_2, s_2) of system in Example 3, we can calculate the following invariants. Here, we only show the few of them that are relevant to our exposition. $DS^{tk_0,tk_1}_{0,s_3} = \{0\}$, $DS^{tk_1,tk_2}_{1,s_2} = \{1\}$, $DS^{tk_2,tk_0}_{2,s_2} = \{1\}$.

We combine these component-state invariants to create two predicates/techniques. These predicates try to establish whether there exist some values N_k, denoting the number of times rule r_k is performed, components can agree on. The first predicate uses these difference sets to derive relationships between N_k variables.

Definition 14. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$. For system state $s = (s_1, \ldots, s_n)$:

$$reach_R(s) \mathrel{\widehat{=}} \exists\, N_1, \ldots, N_m : \mathbb{Z} \bullet \bigwedge_{i \in \{1 \ldots n\}} RC(i, s_i)$$

So, for each component state s_i the Relation Constraint $RC(i, s_i)$ is created as follows.

$$RC(i,s) \mathrel{\widehat{=}} \bigwedge_{k,l \in \Sigma'_i \wedge k \neq l} \begin{cases} True & \text{if } DS^{k,l}_{i,s} = \mathbb{Z} \\ False & \text{if } DS^{k,l}_{i,s} = \emptyset \\ N_k = N_l & \text{if } DS^{k,l}_{i,s} = \{0\} \\ N_k > N_l & \text{for } DS^{k,l}_{i,s} = \{w\} \text{ where } w > 0 \\ N_k < N_l & \text{for } DS^{k,l}_{i,s} = \{w\} \text{ where } w < 0 \end{cases}$$

If $DS^{k,l}_{i,s} = \emptyset$, then state s is unreachable in L'_i. So, it cannot be part of a reachable system state, and we create the unsatisfiable constraint *false*. On the other hand, as $DS^{k,l}_{i,s} = \mathbb{Z}$ gives no information about the value of $N_k - N_l$, it gives rise to the constraint *true*. Finally, if $DS^{k,l}_{i,s}$ holds an integer, we relate N_k and N_l in a simple way according to whether this integer is zero, positive or negative.

If this predicate is satisfiable, it means that components can agree on the number of times shared rules need to be performed to reach the system state being tested. This is not a guarantee the state is reachable but we conservatively assume so. If this predicate is unsatisfiable, however, components cannot agree on such numbers and are, therefore, *unable* to cooperate and reach this system state.

For instance, from the difference sets calculated for system state (s_3, s_2, s_2) of system in Example 3, we can derive the following relation constraints. $RC(0, s_1)$ gives rise to $N_{tk_0} = N_{tk_1}$, $RC(1, s_2)$ to $N_{tk_1} > N_{tk_2}$, and $RC(2, s_2)$ to $N_{tk_2} > N_{tk_0}$. So, we can deduce that $N_{tk_0} = N_{tk_1}$ and $N_{tk_0} > N_{tk_1}$, a contradiction that shows that components cannot agree on the number of times they perform these rules and that $reach_R((s_1, s_2, s_2))$ does not hold.

As our difference sets soundly approximate the behaviour of a component and components must synchronise on shared rules, it follows that, to reach any system state, components must be able to agree particularly on the number of times they engage on shared rules.

Theorem 3. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine with $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For $s \in S$, $reachable(s) \Rightarrow reach_R(s)$.*

Proof. This theorem follows from Theorem 4, as $reach_R$ is a derivation of $reach_D$. □

We point out that $reach_R(s)$ can be decided in polynomial time on the size of the input supercombinator machine.

Lemma 3. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine such that $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $|L_{Max}|$ the size of the largest component in $\mathcal{S}$. For a given state system s of $\mathcal{S}$, we can decide $reach_R(s)$ in time $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2)$.*

Proof. Let $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ be LTS induced by the rule-participation projection $\mathcal{S}_i$, and $DS_i^{k,l}$ computed as per Definition 13. The domain that we use in the CSA framework to calculate $DS_i^{k,l}$ has height $h = \mathcal{O}(1)$. So, from Theorem 1, we can derive that it takes time $\mathcal{O}(|L_i|^2)$ to calculate $DS_i^{k,l}$, and time $\mathcal{O}(|L_i|^2 \cdot |\mathcal{R}|^2)$ to calculate all $DS_i^{k,l}$ for all $k, l \in \Sigma'_i$ where $k \neq l$. So, it takes time $\mathcal{O}(\sum_{i\in\{1\ldots n\}}(|L_i|^2 \cdot |\mathcal{R}|^2))$, which we approximate to $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2))$, to calculate all fixed-points $DS_i^{k,l}$ for $i \in \{1 \ldots n\}$ and $k, l \in \Sigma'_i$ such that $k \neq l$.

The satisfiability of constraints in $reach_R(s)$ can be reduced to checking whether a digraph is cycle free. First of all, we pre-process the constraints in our predicate by soundly removing equations. Each equation $N_k = N_l$ is removed and N_l is replaced by N_k in the remaining constraints. After this pre-processing, the resulting predicate has only inequation constraints and is equisatisfiable. Then, we create a digraph based on this pre-processed predicate. Each variable N_k gives rise to a node and there is an edge from N_k (N_l) to N_l (N_k) if $N_k < N_l$ ($N_k > N_l$, respectively). Constructing this digraph takes time $\mathcal{O}(|\mathcal{R}|^2)$, as there are at most $|\mathcal{R}|$-many N_k variables. Satisfiability of the original formula, thus, amounts to checking whether this digraph is cycle free. This checking can be carried out in time linear on the size of the digraph, so it takes time $\mathcal{O}(|\mathcal{R}|^2)$.

Thus, we have the loose upper-bound of $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2)$ on the time taken to decide $reach_R(s)$ for a fixed system state s. □

3.2.3 Number of rules occurrences – difference consistency. The second predicate creates a system of equations that simply describe the differences captured by our component-state invariants. This predicate generalises $reach_R$.

Definition 15. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$. For system state $s = (s_1, \ldots, s_n)$:

$$reach_D(s) \mathrel{\widehat{=}} \exists\, N_1, \ldots, N_m : \mathbb{Z} \bullet \bigwedge_{i\in\{1\ldots n\}} DC(i, s_i)$$

So, for each component state s_i, Difference Constraint $DC(i, s_i)$ is created as follows.

$$DC(i,s) \mathrel{\widehat{=}} \bigwedge_{k,l\in\Sigma_i \wedge k\neq l} \begin{cases} True & \text{if } DS_{i,s}^{k,l} = \mathbb{Z} \\ False & \text{if } DS_{i,s}^{k,l} = \emptyset \\ N_k - N_l = w & \text{for } DS_{i,s}^{k,l} = \{w\} \end{cases}$$

As for our previous predicate, satisfiability entails that components might be able to agree on the number of times they need to perform shared rules. Thus, while satisfiability approximates reachability, unsatisfiability ensures that components cannot to cooperate to reach this system state.

We point out that $reach_D$ is a strictly more precise approximation than $reach_R$. Any system state that can be shown unreachable by $reach_R$ can also be shown so by $reach_D$, since $reach_D$ generalises $reach_R$. Furthermore, the following example shows that $reach_D$ can show unreachability for some system states that $reach_R$ cannot.

Example 4. Let $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 5 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. We use the name of an event to identify the rule requiring its synchronisation.

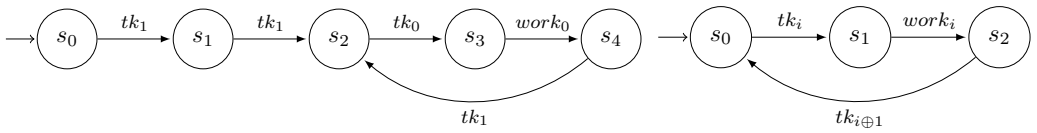

Fig. 5. LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

This system implements a token ring with two tokens where process L_0 has both tokens initially and the events tk_i represent the passage of a token from $L_{i\ominus 1}$ to L_i, where $\ominus$ is subtraction modulo 3. All components can only hold a token at a time with the exception of Component 0 initially.

The system state (s_4, s_2, s_2), for instance, is unreachable as components cannot cooperate to reach it. The system has two tokens that are passed around. In this state, however, all components are full, so they combine to hold 3 tokens, an impossibility. This technique can capture this impossibility. It can derive from $DC(0, s_1)$ that $N_{tk_0} - N_{tk_1} = 1$, from $DC(1, s_2)$ that $N_{tk_1} - N_{tk_2} = 1$, and $DC(2, s_2)$ to $N_{tk_2} - N_{tk_0} = 1$. So, we can deduce that $N_{tk_0} - N_{tk_1} = 1$ and $N_{tk_0} - N_{tk_1} = 2$, a contradiction that shows that components cannot agree on the number of times they perform these rules and that $reach_D((s_4, s_2, s_2))$ does not hold. $reach_D$ can show that any blocked (and locally-blocked) system state is unreachable. On the other hand, $reach_R((s_4, s_2, s_2))$ is unable to do so. For this system state, it gives rise to a consistent set of relations between N_{tk_0}, N_{tk_1}, and N_{tk_2}. ■

The following theorem shows that $reach_D$ over-approximates reachability.

Theorem 4. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine with $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For $s \in S$, $reachable(s) \Rightarrow reach_D(s)$.*

Proof. We assume that $s = (s_1, \ldots, s_n)$ is a reachable system state and show that there exists a collection of values N_k, one for each rule $r_k \in \mathcal{R}$, such that each $DC(i, s_i)$ holds. Let $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ be LTS induced by projection $\mathcal{S}_i$.

If s is reachable, we can assume without lost of generality that $tr = \langle r_{k_1}, \ldots, r_{k_w} \rangle$ is a sequence of rule applications leading the system to s. From tr, we create the collection of values $N_k = tr \downarrow r_k$, where $tr \downarrow r_k$ counts the number of r_k events in tr.

As tr describe a valid behaviour of the system, all L'_i must be able to engage on $tr \upharpoonright \Sigma'_i$ to reach state s_i. Furthermore, for any $r_k \in \Sigma'_i$, it follows that (i) $(tr \upharpoonright \Sigma'_i) \downarrow r_k = N_k$.

From Lemma 4, we know that for any component i and rules $k, l \in \Sigma'_i$, $((tr \upharpoonright \Sigma'_i) \downarrow r_k) - ((tr \upharpoonright \Sigma'_i) \downarrow r_l)$ lies in $DS^{k,l}_{i,s_i}$. Thus, using (i), we can show, that for any component i and rules $k, l \in \Sigma'_i \mid k \neq l$, $N_k - N_l$ lies in $DS^{k,l}_{i,s_i}$. That is, we can show that our collection of values N_k satisfies all constraints $DC(i, s_i)$. □

Lemma 4. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by rule projection $\mathcal{S}_i$, and $DS^{k,l}_{i,s}$ computed as per Definition 13. For any trace tr leading L'_i to its state s, the difference $(tr \downarrow r_k) - (tr \downarrow r_l)$ lies in $DS^{k,l}_{i,s}$,where $tr \downarrow r_k$ counts the number of r_k events in tr.*

Proof. We show that if a fixed-point for values $DS^{k,l}_{i,s}$, where $s \in S'_i$, is reached there cannot be a trace tr of rule events that leads L'_i to s such that $(tr \downarrow r_k) - (tr \downarrow r_l)$ is not in $DS^{k,l}_{i,s}$. We demonstrate that the existence of such a trace tr would lead to a contradiction. We assume without lost of generality that tr is the shortest such trace.

Thanks to equation $DS^{k,l}_{i,\hat{s}'_i} = Init \sqcup DS^{k,l}_{i,\hat{s}'_i}$ and our fixed-point assumption, we know that $0 \in DS^{k,l}_{i,\hat{s}}$. Clearly, then, tr cannot be $\langle\rangle$ as $tr = \langle\rangle$ only leads to $\hat{s}'_i$ and $(\langle\rangle \downarrow r_k) - (\langle\rangle \downarrow r_l) = 0$ is indeed in $DS^{k,l}_{i,\hat{s}'_i}$.

Thus, we can safely assume that tr is of the form $tr'^\frown\langle rk\rangle$, where $\hat{s}'_i \xrightarrow{tr'} s' \xrightarrow{r_k} s$. Moreover, thanks to our assumption that tr is the shortest trace, we know that $(tr' \downarrow r_k) - (tr' \downarrow r_l) \in DS^{k,l}_{i,s'}$. From this fact, we can deduce that the equation $DS^{k,l}_{i,s} = T_{r_k}(DS^{k,l}_{i,s'}) \sqcup DS^{k,l}_{i,s}$, corresponding to transition $s' \xrightarrow{r_k} s$, does not hold, contradicting our fixed-point assumption. Note that while the value $(tr \downarrow r_k) - (tr \downarrow r_l)$ is in $T_{r_k}(DS^{k,l}_{i,s'}) \sqcup DS^{k,l}_{i,s}$, it is not a member of $DS^{k,l}_{i,s}$. □

Next, we show that $reach_D$ can be checked in polynomial time on the size of the input supercombinator machine.

Lemma 5. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine such that $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $|L_{Max}|$ the size of the largest component in $\mathcal{S}$. For a given state system s of $\mathcal{S}$, we can decide $reach_D(s)$ in time $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2 + |\mathcal{R}|^3)$.*

Proof. Let $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$, be the LTS induced by the rule-participation projection of $\mathcal{S}_i$, and $DS^{k,j}_i$ computed as per Definition 13. The domain that we use in the CSA framework to calculate $DS^{k,l}_i$ has height $h = \mathcal{O}(1)$. So, from Theorem 1, we can derive that it takes time $\mathcal{O}(|L_i|^2)$ to calculate $DS^{k,l}_i$, and time $\mathcal{O}(|L_i|^2 \cdot |\mathcal{R}|^2)$ to calculate all $DS^{k,l}_i$ for all $k, l \in \Sigma'_i$ where $k \neq l$. So, it takes time $\mathcal{O}(\sum_{i\in\{1\ldots n\}}(|L_i|^2 \cdot |\mathcal{R}|^2))$, which we approximate to $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2))$, to calculate all fixed-points $DS^{k,l}_i$ for $i \in \{1 \ldots n\}$ and $k, l \in \Sigma'_i$ such that $k \neq l$.

The constraints in $reach_D(s)$ give rise to a formula in a known fragment of linear integer arithmetic called *difference logic* [51]. Formulas in this fragment can be decided in polynomial time through a reduction to checking negative cycles in a digraph with edges annotated with integer weights. This weighted digraph has a node for each variable N_k in our predicate, and each difference equation $N_k - N_l = w$ gives rise to a pair of edges: one from N_k to N_l with weight w and another from N_l to N_k with $-w$. Given this digraph, we can use Bellman-Ford algorithm [16, 33, 44] to check whether it has a cycle with a negative sum of weights. For a digraph with nodes V and edges E, this algorithm takes $\mathcal{O}(|V| \cdot |E|)$ time. There can be as many as $|\mathcal{R}|$ variables N_k. So, it takes $\mathcal{O}(|\mathcal{R}|^2)$ to construct this digraph from our difference constraints. The resulting digraph has $|V| = \mathcal{O}(|\mathcal{R}|)$ and $|E| = \mathcal{O}(|\mathcal{R}|^2)$, placing an upper-bound of $\mathcal{O}(|\mathcal{R}|^3)$ on the time this algorithm needs to check this formula.

Thus, we have the loose upper-bound of $\mathcal{O}(n \cdot |L_{Max}|^2 \cdot |\mathcal{R}|^2 + |\mathcal{R}|^3)$ on the time taken to decide $reach_D(s)$ for a fixed system state s. □

3.3 Rule abstraction

We can extend and improve these techniques by carrying out some abstractions.

We improve our techniques by ignoring *single-participant rules*. The rules that require the participation of a single component do not contribute to the sort of inconsistency on how components collaborate our techniques look for. So, they do not influence at all in the sort of reachability analysis they carry out. To make the $reach_S$ technique ignore these rules, we change Definition 10 to make $T_{r_k}(v) = v$ whenever r_k is a single-participant rule. To make $reach_D$ and $reach_R$ ignore these rules, we change constraints DC in Definition 15 and RC in Definition 14. We modify both constraints so that they only create conjuncts for

pairs k, l if both rules r_k and r_l are not single-participant. This improvement is sound since single-participant rules do not interfere with the sort of consistency checking between the behaviour of components carried out by our techniques.

In the following, we propose two abstractions that can be used to capture component invariants that are beyond the capabilities of our original techniques. These abstractions partition rules of a supercombinator machine into equivalence classes. So, rules in the same partition are identified and treated as the same. We use $[r_k]$ to denote the partition rule r_k belongs to and $min[r_k]$ to denote the *representative* of this partition, namely, the rule with the smallest index k in $[r_k]$.

3.3.1 Data abstraction. We can achieve a sort of data abstraction for our techniques as follows. Intuitively, the application of a rule can be seen as a communication taking place between participants, whereas a set of rules involving the same exact participants might be seen as a set of possible values they can communicate. With this view in mind, if we identify rules with the same participants, we are abstracting away these values and focusing on the fact a communication occurred between these participants. We use the following projection to capture this kind of data abstraction.

Definition 16. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine. We use $r_k = (e, a)$ to denote that rule (e, a) is identified by k. This abstraction uses the following partitioning $[r_k] \mathrel{\widehat{=}} \{r'_k \mid r'_k \in \{r_1, \ldots, r_m\} \wedge pts(r'_k) = pts(r_k)\})$, where $pts(r_k) \mathrel{\widehat{=}} \{i \mid i \in \{1 \ldots n\} \wedge e_i \neq -\}$ gives the participants of $r_k = (e, a)$. So, the data-abstraction rule-participation projection of $\mathcal{S}$ on component i is given by $\mathcal{S}_i^{DA} = (\langle L_i \rangle, \{((e_i), min[r_k]) \mid r_k = (e, a) \in \mathcal{R} \bullet e_i \neq -\})$.

We integrate this abstraction into our techniques by using this data-abstraction projection, instead of the original rule-participation projection, to calculate component-state invariants. This modification gives rise to the following data-abstract counterparts to our original predicates: $reach_S^{DA}(s)$, $reach_D^{DA}(s)$, and $reach_R^{DA}(s)$.

This abstraction can capture some invariants, both difference sets and suffixes, that are beyond the capabilities of our original techniques. We illustrate such an invariant with the following example.

Example 5. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 6 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. We use the name of an event to identify the rule requiring its synchronisation. This system implements a token ring where process L_0 has the token initially. The events $tk_{i,v}$ represent the passage of a token from $L_{i\ominus 1}$ to L_i, where $\ominus$ is subtraction modulo 3, and $v \in \{0, 1\}$ denotes some value that annotates the token. In our example, for simplicity, our components do not make use of it but they could use it to perform different kinds of work, for instance.

The invariants that these components respect are of the form: $(N_{tk_{i,0}} + N_{tk_{i,1}}) - (N_{tk_{i\oplus 1,0}} + N_{tk_{i\oplus 1,1}})$. This sort of invariant cannot be captured by our original techniques $reach_R$ or $reach_D$ – they only individually relate N_{r_k} variables – and since these variables are not individually related, these techniques capture meaningless $\mathbb{Z}$s as difference sets. So, our original techniques give rise to the pointless reachability approximations $reach_R(s) = reach_D(s) = true$.

On the other hand, our data abstraction allows $reach_R^{DA}$ and $reach_D^{DA}$ to capture these *sum* invariants. This abstraction identifies rules $tk_{i,0}$ and $tk_{i,1}$. So, for each component i, it calculates $DS_{i,s_i}^{[tk_{i,0}],[tk_{i\oplus 1,0}]}$, the difference set for $N_{[tk_{i,0}]} - N_{[tk_{i\oplus 1,0}]}$. Note that $N_{[tk_{i,0}]}$

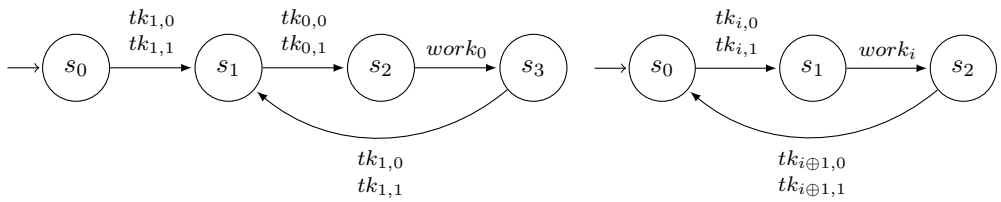

Fig. 6. LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

counts how many times rules in $\{tk_{i,0}, tk_{i,1}\}$ have to be performed to reach s_i, i.e. the sum of times rules in this set are performed. Broadly speaking, our data abstraction can be seen as collapsing transitions involving $tk_{i,0}$ and $tk_{i,1}$ into one. So, the analysis of this system becomes similar to the analysis we carry out in Example 3.

For blocked state (s_3, s_2, s_2), for instance, our abstraction gives rise to difference sets $DS_{0,s_3}^{[tk_{0,0}],[tk_{1,0}]} = 0$, $DS_{1,s_2}^{[tk_{1,0}],[tk_{2,0}]} = 1$, and $DS_{1,s_2}^{[tk_{2,0}],[tk_{0,0}]} = 1$. So, technique $reach_R^{DA}$ ($reach_D^{DA}$) can derive that $N_{[tk_{1,0}]} = N_{[tk_{0,0}]}$ ($N_{[tk_{1,0}]} - N_{[tk_{0,0}]} = 0$) and $N_{[tk_{1,0}]} > N_{[tk_{0,0}]}$ ($N_{[tk_{1,0}]} - N_{[tk_{0,0}]} = 2$), a contradiction that shows that $reach_R^{DA}((s_3, s_2, s_2))$ (respectively, $reach_D^{DA}((s_3, s_2, s_2))$) does not hold. Therefore, this blocked state is unreachable as components cannot cooperate to reach it. In fact, this abstraction allows our techniques to show all blocked and locally-blocked system states unreachable. ■

The same idea we use in this example can be applied to demonstrate that $reach_S^{DA}$ captures suffixes different from the ones $reach_S$ finds. As for this example, we can modify the system in Example 2 to have events $ring_{i,v}$, where $v \in \{0, 1\}$, instead of their original $ring_i$ events. For this modified system, $reach_S^{DA}$ captures useful suffixes and a reachability approximation that proves it deadlock and local-deadlock free, whereas $reach_S$ does not.

We point out that all levels of abstractions that we propose in this work are *incomparable* in the sense that they capture different invariants and, hence, the set of systems they prove (local-)deadlock free are incomparable.

The same formal argument that we originally used to show that our original techniques over-approximate reachability can be slightly modified to show that their abstract versions also do so. Furthermore, we point out that the original polynomial bounds on the time needed to decide whether our predicates hold are also bounds for their respective abstract counterparts.

Theorem 5. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine with $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For $s \in S$, $reachable(s) \Rightarrow reach_x^{DA}(s)$, where x can be S, D, or R.*

3.3.2 Component-specific abstractions. For the $reach_D$ technique, we can propose further sorts of rule partitioning that can capture other relational invariants of components. In some cases, the relational invariant a component respects is not given by how individual rules relate or by how rules with the same participants relate. It might even be the case that different components require different rule partitionings. So, we introduce a new version of this technique's predicate, $reach_D^{CA}$, that allows each component to have its own and customised partitioning.

Definition 17. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine, and $[\cdot]_1, \ldots, [\cdot]_n$ rule partitionings where $[\cdot]_i$ partition the rules component i participate in, i.e. $\{r_k \mid r_k = (e, a) \in \mathcal{R} \wedge e_i \neq -\}$. For system state $s = (s_1, \ldots, s_n)$:

$$reach_D^{CA}(s) \mathrel{\widehat{=}} \exists\, N_1, \ldots, N_m : \mathbb{Z} \bullet \bigwedge_{i \in \{1 \ldots n\}} DC(i, s_i)$$

So, for each component state s_i, Difference Constraint $DC(i, s_i)$ is created as follows.

$$DC(i,s) \mathrel{\widehat{=}} \bigwedge_{r_k, r_l \in \Sigma_i \wedge k \neq l} \begin{cases} True & \text{if } DS_{i,s}^{k,l} = \mathbb{Z} \\ False & \text{if } DS_{i,s}^{k,l} = \emptyset \\ (\sum_{r'_k \in [r_k]_i} N_{k'}) - (\sum_{r'_l \in [r_l]_i} N_{l'}) = w & \text{for } DS_{i,s}^{k,l} = \{w\} \end{cases}$$

The difference sets $DS_{i,s}^{k,l}$ of component i are calculated on the LTS induced by the following projection of $\mathcal{S}$ on i: $(\langle L_i \rangle, \{((e_i), min[r_k]_i) \mid r_k = (e, a) \in \mathcal{R} \bullet e_i \neq -\})$. This projection depicts the behaviour of this component in terms of the rule it participates and considering the partitioning $[\cdot]_i$.

This predicate can be fitted to use any partitioning for components, and different partitionings can capture different relations between variables N_k. In this work, however, we use it with the following *fixed* component partitioning. We chose this partitioning because it is fairly simple and captures some interesting relational invariants we found in our analyses of systems. So, when we use $reach_D^{CA}$, we mean this technique with the following component partitioning.

Definition 18. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \{r_1, \ldots, r_m\})$ be a supercombinator machine, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ the LTS induced by the rule-participation projection $\mathcal{S}_i$. We define partitioning $[\cdot]_i$ as the finest partitioning such that: if there exists $s, s' \in S'_i$ and $r_k, r_{k'} \in \Sigma'_i$ where $(s, r_k, s') \in \Delta'_i$ and $(s, r_{k'}, s') \in \Delta'_i$, then $[r_k]_i = [r_{k'}]_i$. This partitioning identifies any two rules giving rise to the same component transition. This partitioning can be efficiently calculated using *partition refinement* approaches [65].

Next, we illustrate how this technique can capture invariants that cannot be captured neither by $reach_D$ nor $reach_D^{DA}$.

Example 6. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 7 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. We use an event name to identify the rule requiring its synchronisation. This system implements a token network where process L_0 has the token initially and the events $tk_{i,j}$ represent the passage of a token from L_i to L_j.

For each component the sum of output (rule) events is relate to the sum of input events. For instance, for Component 0, $N_{tk_{0,1}} + N_{tk_{0,2}}$ is related to $N_{tk_{1,0}} + N_{tk_{2,0}}$. Since there are multiple output and input rule events and these do not involve the same participants both $reach_D$ and $reach_D^{DA}$ are unable to capture meaningful difference sets. So, they give rise to $reach_D(s) = reach_D^{DA}(s) = true$.

On the other hand, our component-specific abstraction allows $reach_D^{CA}$ to capture these invariants. The partitioning that we propose identify input and output rule events in a way that we capture the exact invariant needed. For this system, the partitioning is as follows.

- $[\cdot]_0$ is given by $\{r_{tk_{1,0}}, r_{tk_{2,0}}\}, \{r_{tk_{0,1}}, r_{tk_{0,2}}\}$
- $[\cdot]_1$ is given by $\{r_{tk_{0,1}}, r_{tk_{2,1}}\}, \{r_{tk_{1,0}}, r_{tk_{1,2}}\}$
- $[\cdot]_2$ is given by $\{r_{tk_{0,2}}, r_{tk_{1,2}}\}, \{r_{tk_{2,0}}, r_{tk_{2,1}}\}$

Our abstraction creates difference sets $DS_{0,s_3}^{[tk_{0,1}],[tk_{1,0}]} = \{0\}$, $DS_{1,s_2}^{[tk_{1,0}],[tk_{0,1}]} = \{1\}$, and $DS_{1,s_2}^{[tk_{2,0}],[tk_{0,2}]} = \{1\}$ for blocked state (s_3, s_2, s_2). So, technique $reach_D^{CA}$ can derive from these sets: $(N_{tk_{1,0}} + N_{tk_{2,0}}) - (N_{tk_{0,2}} + N_{tk_{0,1}}) = 0$, $(N_{tk_{0,1}} + N_{tk_{2,1}}) - (N_{tk_{1,0}} + N_{tk_{1,2}}) = 1$,

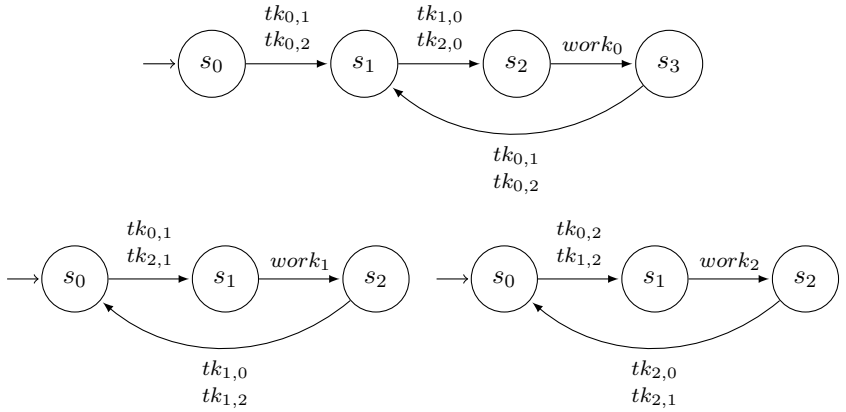

Fig. 7. LTSs of components L_0, L_1, and L_2, respectively.

and $(N_{tk_{0,2}} + N_{tk_{1,2}}) - (N_{tk_{2,0}} + N_{tk_{2,1}}) = 1$, respectively. From these, we can derive that $(N_{tk_{1,0}} + N_{tk_{2,0}}) - (N_{tk_{0,2}} + N_{tk_{0,1}}) = 0$ and that $(N_{tk_{1,0}} + N_{tk_{2,0}}) - (N_{tk_{0,2}} + N_{tk_{0,1}}) = 2$ (by adding the second and third equations, respectively), a contradiction that shows that $reach_D^{CA}((s_3, s_2, s_2))$ does not hold. Therefore, this blocked state is unreachable as components cannot cooperate to reach it. In fact, this abstraction can show all blocked and locally-blocked system states unreachable. ■

An argument similar to the one we used to show that $reach_D$ soundly approximates reachability can be used to show that this new predicate over-approximates reachability. Unlike our previous predicates, it does not seem that, for a given state, this predicate can be checked in polynomial time. In fact, difference sets can be calculated in polynomial time but we end up with a linear-integer-arithmetic formula and checking satisfiability for such a formula constitutes a *NP*-complete problem.

Theorem 6. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine with $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For $s \in S$, $reachable(s) \Rightarrow reach_D^{CA}(s)$.*

We cannot trivially extend this component-based abstraction for our suffix-based technique. The reason is that while our difference sets can easily accommodate different partitionings using these sums of variables, for the suffix-based technique, we would need to replace a suffix by a tree-like structure that captures the recent behaviour of a system in terms of such partitionings. For such a tree-like structure, we would not be able to use our occurrence information, in the simple way we do, to uniquely identify system-wide occurrences of rules and to compare components' behaviour.

3.4 Discussion

The imprecision for the sort of technique we propose stems from the use of component-state invariants. These invariants can cause imprecision in two ways. Firstly, the information that we capture might not be what is required to show unreachability. Note that our component-state invariants only give a partial account of components' behaviour; there are unreachable system states for which components might agree on occurrence suffixes or on the number of times they need to perform shared rules to get there. Secondly, the process of calculating these invariants, namely, of compactly summarising a set of behaviours, might lead to the

discarding of behavioural information that could be relevant in showing a system state unreachable. For instance, assume that a system state is unreachable because components cannot cooperate on the last rule they perform before reaching this state and each component can perform two different last rules to reach this state (also, assume these last rules do not involve the same participants so our data abstraction mechanism would not identify them). In this case, the techniques $reach_S$ and $reach_S^{DA}$ would summarise the behaviour of each component with the empty sequence, so it would imprecisely assume that such a system state is reachable. A similar sort of imprecision would happen for $reach_R$ and $reach_D$ and its abstract counterparts if some system state was unreachable due to components' difference sets having two values instead of the single valued sets our domain allows. In such cases, our component-state invariant approximates this set of two values to the entire set of integers.

Despite being imprecise, our approximations are precise enough to show some interesting properties of distributed systems employing common interaction paradigms. The use of both recent behaviour and relational invariants to characterise component states and show they cannot interact to reach some undesired system state is fairly common. For instance, relational invariants are commonly employed by to characterise token mechanisms and that components can only be so-far apart in terms of rounds of communications, whereas some other methods use the recent behaviour of components to show that components cannot be simultaneously blocked [37, 52, 58, 74]. Examples 2, 3, 4, 5, and 6 all illustrate some common mechanisms that our techniques can capture.

4 PAIRSTATIC: PAIRWISE REACHABILITY MEETS GLOBAL INVARIANTS

In this section, we propose *PairStatic*: an approximate framework for checking deadlock and local-deadlock freedom that extends the *Pair* framework with the approximation techniques proposed in this paper. Unlike *Pair*, this new framework can captures some global system invariants. We propose two versions for this framework. One uses predicate $reach_D$ and its abstractions and the other $reach_R$ instead. These two versions are characterised by the following definition of a (local-)deadlock *candidates*.

Definition 19. Let $\mathcal{S}$ be a supercombinator machine. A system state s is:

- a $PairStatic_R$ *candidate* if and on if $candidate_R(s)$ holds;
- a $PairStatic_D$ *candidate* if and on if $candidate_D(s)$ holds;
- a $PairStatic_R$ *local candidate* if and on if $local\_candidate_R(s)$ holds;
- a $PairStatic_D$ *local candidate* if and on if $local\_candidate_D(s)$ holds.
  - $candidate_R(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach_S(s) \wedge reach_R(s) \wedge reach_S^{DA}(s) \wedge reach_S^{DA}(s)$
  - $candidate_D(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach_S(s) \wedge reach_D(s) \wedge reach_S^{DA}(s) \wedge reach_D^{DA}(s) \wedge reach_D^{CA}(s)$
  - $local\_candidate_x(s)$ is the obtained by replacing $blocked(s)$ by $locally\text{-}blocked(s)$ in the corresponding predicate $candidate_x(s)$.

Note that as a blocked state is also locally-blocked, a $PairStatic_x$ candidate must also be a $PairStatic_x$ local candidate. Therefore, a system that is free of $PairStatic_x$ *local* candidates must also be free of $PairStatic_x$ candidates; although the converse does not always hold.

The reason for proposing two different versions for our framework relates to efficiency and simplicity in implementation. We initially created and studied the simplified version involving $reach_R$ and its abstraction because it fitted easily into our SAT-based approach and it captured some relevant invariants. After this initial study, we created our generalised version, using predicate $reach_D$, which we implemented using a SMT-based translation. Note that

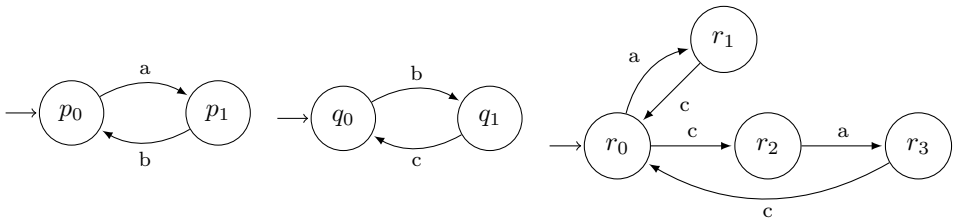

Fig. 8. LTSs of components L_1, L_2 and L_3, respectively.

our $PairStatic_D$ framework is strictly more precise than $PairStatic_R$ since approximations $reach_D$ and $reach_D^{DA}$ are more precise than $reach_R$ and $reach_R^{DA}$, respectively.

Unsurprisingly, given that the techniques we propose over-approximate reachability, they can be soundly used to check that a system is deadlock and local-deadlock free[1].

Theorem 7. *If a supercombinator machine is $PairStatic_x$ candidate free, where x is R or D, then it must also be deadlock free. The same implication holds for $PairStatic_x$-local-candidate freedom and local-deadlock freedom.*

4.1 Precision and complexity of PairStatic

This new framework is clearly more precise than *Pair*, but it remains imprecise: a (locally-)blocked system state can be unreachable and yet meet all reachability tests/predicates we proposed. As for 2-reachability, none of these new tests are precise enough to show, for instance, that the single (locally-)blocked system state in Example 7 is unreachable. So, like *Pair*, *PairStatic* is unable to show that the system in that example is (local-)deadlock free. Nevertheless, by conjoining these new tests, we tighten the state space analysed. Observe that it only takes one failed reachability test, out of all approximations proposed, to consider a system state unreachable.

Example 7. Let $\mathcal{S} = (\langle L_1, L_2, L_3\rangle, \mathcal{R})$ be the supercombinator machine such that the components are described graphically in Figure 8 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) passes all our reachability tests (approximations) and is blocked (and locally-blocked), but it is not reachable. Thus, it constitutes a *PairStatic* (local) candidate but not a (local) deadlock. ■

Unlike 2-reachability, the sort of combination of component invariants that we propose in our new reachability tests allows for *global analysis*. By checking consistency between component-state invariants for *all* components, *PairStatic* can capture some global system invariants. Examples 2, 3, 4, 5 and 6 all illustrate global invariants that cannot be captured by local analysis. So, while *PairStatic* can capture these global invariants and show that the systems in these examples are (local-)deadlock free, *Pair* cannot due to its use of pure local analysis.

In the following, we introduce a few examples to illustrate and discuss some mechanisms that distributed systems implement to avoid (locally-)blocked states and that we can capture with *PairStatic*. Furthermore, we use these examples to compare our framework with Martin's *FSDD* and *CSDD*; two extensions of his *SDD* (State Dependency Digraph) framework [58]. SDD constructs a dependency digraph based on the analysis of pair of components, which is

[1]The proof that 2-reachable approximates reachability can be found in [6].

later checked for absence of cycles; this framework characterises a deadlock candidate as a cycle of dependencies amongst components. It turns out that this framework is very efficient but its candidate characterisation makes this approach fairly imprecise. To make it more precise, Martin extended the SDD by adding extra reachability information to edges of the dependency digraph. While CSDD adds information about the number of *cycles* components engaged on, FSDD adds information about the most recent interaction of components. CSDD considers the point when a component returns to is initial state to be the completion of a cycle of interactions. Then, it tries to establish whether, for a given cycle, each component has performed more cycles than the next. If so, we have an inconsistency/impossibility. Similarly, FSDD establishes whether, in a cycle, each component has interacted more recently with its predecessor. If so, the cycle leads to an impossibility regarding the order in which these interactions took place. These two methods were proposed to prove deadlock for classes of systems with cyclically-interacting components, which usually implement some global mechanism to avoid (local-)deadlocks. In fact, the property that they try to prove is stronger than local-deadlock freedom. CSDD can tackle some systolic-array-like systems, whereas FSDD was designed to tackle non-fillable systems. Despite being more precise, they inherit some of SDD's sources of imprecision.

The first mechanism we discuss, proposed in [23], prevents systems from having all their components simultaneously full. We call this the *non-fillable* mechanism. In a non-fillable system, components are disposed in a ring and they can exchange and store messages. A component implements this mechanism if the last action that fills its finite storage space is an incoming message from its predecessor in the ring. If components of a system implement this mechanism, they cannot be simultaneously full as components cannot agree on an order in which they perform this last action. This mechanism and contradiction are illustrated in Example 2. In the following, we introduce a modified version of Example 2 that can be shown (local-)deadlock free by *PairStatic* but not by FSDD. We point out that [72] introduces an approach to systematically transforming a non-fillable ring into a (local-)deadlock-free system with an arbitrary topology, and *PairStatic* can capture this sort of invariant even for such an extension.

Example 8. Let $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ be the supercombinator machine with L_i defined in Figure 9 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. This supercombinator machine adds the event chk_i to model component i's action of checking a message's integrity to the system in Example 2.

Both FSDD and *PairStatic* can show (local-)deadlock freedom for Example 2 but only *PairStatic* can show (local-)deadlock freedom for this system. *PairStatic* can show that this system is (local-)deadlock free using a systematic argument that is very similar to the one that we present for Example 2. FSDD and $reach_S$ were designed to find the same sort of contradiction on the recent behaviour of components. While FSDD tries to show that components are unable to cooperate using each component's last action, $reach_S$ uses a suffix/sequence of rule events/actions. So, while in Example 2 the contradiction arises from each component's last action, in this example, thanks to the addition of event chk_i, it arises due to the second-to-last action. So, only our framework can show (local-)deadlock freedom for this example.

This example presents a non-fillable system where components can store up to two messages. *PairStatic*, however, can show (local-)deadlock freedom for versions of this system where components can store any finite number of messages. Secondly, event chk_i could be replaced by some more complex behaviour, such as some interaction with other components,

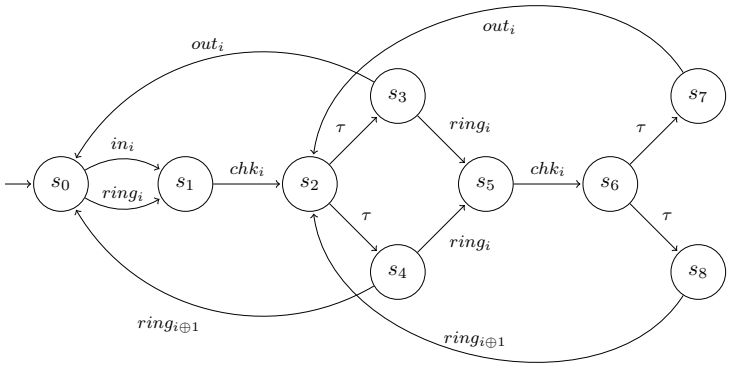

Fig. 9. LTS of component L_i where $\oplus$ represents addition modulo 3.

and as long as the the actions leading to a contradiction can be captured by our suffixes, *PairStatic* should be able to show (local-)deadlock freedom for it. ■

The next mechanism we discuss, proposed in [74, 76], can be used to construct *systolic-array*-like systems. In a systolic-array, components cyclically interact with their neighbours so they collaborate to perform some task. Given a partial order on system rules, a system implements the *cyclic ordering* mechanism if components behave cyclically by participating in rules following this order. For such a system, components cannot reach (locally-)blocked states because they are unable to agree on the number of times they participate in shared rules to reach it. Unlike our non-fillable mechanism, this one does not impose any restriction, a priori, on the topology of the system. The following example illustrates this mechanism. It introduces a system that can be tackled by *PairStatic* but not by CSDD.

Example 9. Let $\mathcal{S} = (\langle L_{1,1}, L_{1,2}, L_{2,1}, L_{2,2}\rangle, \mathcal{R})$ be the supercombinator machine with components $L_{i,j}$ defined in Figure 10 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. This system implements a grid-like systolic array. Each component behave cyclically as follows. It interacts with its left and up neighbours via $h_{i,j}$ and $v_{i,j}$, respectively. Then, it does some processing using either event $p_{i,j}$ or $p'_{i,j}$. Finally, it communicates with its right and down neighbours via $h_{i+1,j}$ and $v_{i,j+1}$, respectively. Each component initially (non-deterministically) chooses which type of processing it will do: either $p_{i,j}$ or $p'_{i,j}$. Note that every neighbouring pair of processes in this network can reach the complete Cartesian product of states. Therefore, *Pair* makes no actual restriction here.

PairStatic can show that this system is (local-)deadlock free using a systematic analysis that is very similar to the one that we present for Example 3, whereas CSDD cannot. While CSDD captures that a component completed a cycle when it transitions back to its initial state, $reach_R$ does so by using relations between number of times components perform shared rules. In this example, cycles are not completed when a component reaches back its initial state but when it reaches state s_1 or s_6. So, while $reach_R$ can capture the cyclic behaviour of this system and use it to prove (local-)deadlock freedom, CSDD cannot. Note that event $p_{i,j}$ could be replaced by some complex behaviour and, as long as it conforms to this mechanism, *PairStatic* should handle it. ■

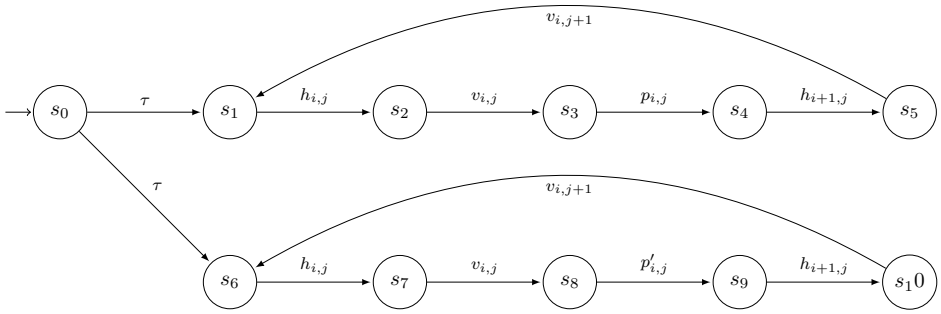

Fig. 10. Sketch of LTS for components $L_{i,j}$.

These two mechanisms are also formally presented and discussed in [58] and [73].

Lastly, we discuss a *token-ring* mechanism that allows for tokens to be exchanged between components. In a token-ring system, components are arranged in a ring and they can exchange (and store) tokens. A system conforms to this mechanism, if components behave, cyclically, by acquiring tokens from their predecessor and passing them along to their successor in the ring (they might also carry out some additional individual behaviour). Also, for a system with m token capacity (i.e. the sum of all components' storage space), it must initially hold between 1 and $m-1$ tokens. If components of a system implement this mechanism, they can neither be simultaneously full nor empty as components cannot agree on the number of times they exchange tokens. This mechanism and contradiction are illustrated in Examples 3, 4 and 5. In the following, we introduce a more complex example of a system implementing this mechanism that can be tackled by *PairStatic* but not by CSDD or FSDD. The same idea proposed in [72] to convert a non-fillable rings into a system with arbitrary topology can be used to the same effect on a system implementing our token-ring mechanism, and *PairStatic* should be able to capture this extension. This idea can be used to, for instance, create the fully-connected (local-)deadlock-free system in Example 6.

Example 10. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_i defined in Figure 11 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. This system implements a token ring where process L_0 has the token initially and the events tk_i represents the passage of a token from $L_{i\ominus 1}$ to L_i, where $\ominus$ is subtraction modulo 3.

Again, *PairStatic* can show that this system is (local-)deadlock free using a systematic analysis that is very similar to the one that we carried out for Example 5. The cycle-counting analysis CSDD carries out, however, does not capture the sort of token conservation invariant that is required to show that this system is (local-)deadlock free. This system can be modified so that components can hold multiple messages and so that the system is neither empty nor full of tokens initially, and *PairStatic* could still check it (local-)deadlock free. ■

Generally speaking, our strategy should prove (local-)deadlock freedom if (locally-)blocked states induce the sort of inconsistency our predicates are tailored to find. Another point worth making is that, unlike FSDD and CSDD, we combine/conjoin our reachability approximations. So, if the (locally-)blocked states of a system are divided into states that are unreachable because of components' suffixes incompatibility and states that are unreachable because of components' inability to agree on the number of times they need to perform shared rules,

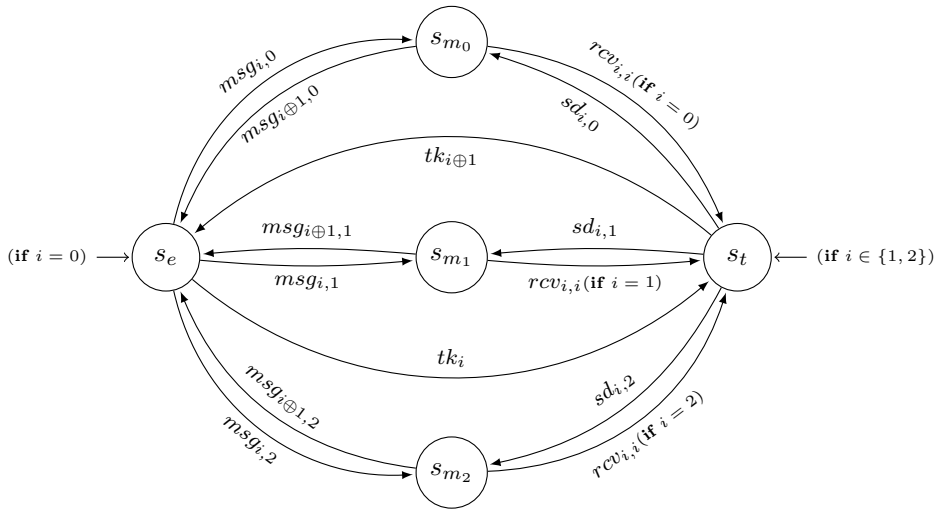

Fig. 11. Sketch of LTS for components L_i.

our combination of approximations allow *PairStatic* to demonstrate that all these states are unreachable.

The increase in precision we gain by conjoin these new predicates to our *Pair* candidate definition comes at a reasonable price. By conjoining reachability approximations that can be decided in polynomial time for a fixed system state to our *Pair* candidate definition, we end up with a detection problem that is *NP*-complete. Note, however, that we could not find an algorithm to decide $reach_D^{CA}$ for a given state in polynomial time. Hence, for $candidate_D$, we only prove a lower bound/hardness result. We also only prove a hardness result for the local-candidates counterpart problems. These results also justify our use of SAT and SMT solving in tackling these problems.

Theorem 8. *The problem of deciding whether a supercombinator machine has a system state satisfying* $candidate_R$ *is* NP*-complete, whereas deciding the same problem for* $candidate_D$ *is* NP*-hard. The counterpart problems for local candidates are also* NP*-hard.*

Proof. The problem of detecting a $candidate_R$ state is a member of *NP* since for a fixed system state, all our reachability-approximating predicates can be decided in polynomial time thanks to Lemmas 2, 5, and 3; $reach_2$ can be decided by examining the state space of pairs of components. *NP*-hardness for this problem and detecting $candidate_D$ states follows from the following corollary.

Corollary 1 (Corollary 1 in [6])**.** *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.*

In the proof to Theorem 2 in [6], we present a construction that, in polynomial time, transforms a given supercombinator machine $\mathcal{S}$ into a machine $\mathcal{T}'(\mathcal{S})$ for which the *blocked* predicate for $\mathcal{S}$ coincides with the *locally-blocked* predicate for $\mathcal{T}'(\mathcal{S})$. Furthermore, this translation has no impact in any of the reachability approximations that we present here. So, our approximations coincide for the two machines. Therefore, this construction can be used to

reduce the $candidate_R$-state ($candidate_D$-state) detection problem to the $local\_candidate_R$-state ($local\_candidate_D$-state) detection problem, proving the latter *NP*-hard. □

4.2 PairStatic-candidate detection via SAT/SMT solving

We built upon our SAT-checking approach proposed for detecting *Pair* candidates in [6] to create an efficient implementation for *PairStatic*. We implement our framework using SAT formulas $PairStatic_R$ and $LocalPairStatic_R$, and SMT formulas $PairStatic_D$ and $LocalPairStatic_D$. While $PairStatic_R$ ($LocalPairStatic_R$) captures states that satisfy $candidate_R$ ($local\_candidate_R$), $PairStatic_D$ ($LocalPairStatic_D$) detects states satisfying $candidate_D$ ($local\_candidate_D$). We choose to use the theory of linear integer arithmetic to encode $PairStatic_D$ and $LocalPairStatic_D$ as it is more convenient and SMT solvers tend to be efficient in handling such formulas. So, we encode the search for a (local-)deadlock candidate as a satisfiability problem to be later checked by a SAT/SMT solver. For the remainder of this section, let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $\mathcal{S}_i$ the projection of $\mathcal{S}$ on component i, and $L'_i = (S'_i, \Sigma'_i, \Delta'_i, \hat{s}'_i)$ its induced LTS.

We use boolean variables $st_{i,s}$ to represent state s of component i. Our formulas are constructed so the combination of component-state variables assigned to true in a satisfying assignment forms an appropriate (local-)deadlock candidate. These formulas conjoin a sub-formula for each predicate in our candidate definitions; each sub-formula holds for a combination of component states that satisfies the corresponding predicate. $LocalPairStatic_x$ is constructed by substituting *Blocked* by *LocallyBlocked* in the definition of $PairStatic_x$.

$$\begin{aligned} PairStatic_R \mathrel{\widehat{=}} \quad & State \wedge Blocked \wedge Reach_2 \wedge Reach_S \wedge Reach_S^{DA} \\ & \wedge Reach_R \wedge Reach_R^{DA} \end{aligned}$$

$$\begin{aligned} PairStatic_D \mathrel{\widehat{=}} \quad & State \wedge Blocked \wedge Reach_2 \wedge Reach'_S \wedge Reach_S^{DA\prime} \\ & \wedge Reach_D \wedge Reach_D^{DA} \wedge Reach_D^{CA} \end{aligned}$$

Our formulas reuse sub-formulas *State*, *Blocked*, *LocallyBlocked* and $Reach_2$ proposed in [6]. To make this paper self-contained, we reintroduce (slightly simplified versions of) them here. 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.

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

The sub-formula *Blocked* captures the *blocked* predicate. Thanks to triple disjointness, we can capture whether a system state is blocked or locally 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 rely on $s \longrightarrow_i$ and $(s, s') \longrightarrow_{i,j}$; they denote that the system can transition if component i is in state s and if components i and j are in states s and s', respectively. Formally, $s \longrightarrow_i$ ($(s, s') \longrightarrow_{i,j}$) holds iff there exists a rule $(e_1, \ldots, e_n) \in \mathcal{R}$ such that $e_i \neq -$ (and $e_j \neq -$) and all other $e_k = -$ and it can be triggered when component i (and j) is in $s \in S_i$ (are in $(s, s') \in S_i \times S_j$). So, this encoding forbids combinations of component states that makes the system progress.

$$Blocked \mathrel{\widehat{=}} \Big(\bigwedge_{\substack{i \in \{1 \ldots n\} \\ \wedge s \longrightarrow_i}} \neg st_{i,s} \Big) \wedge \Big(\bigwedge_{\substack{i,j \in \{1 \ldots n\} \wedge i \neq j \\ \wedge (s,s') \longrightarrow_{i,j}}} \neg st_{i,s} \vee \neg st_{j,s'} \Big)$$

To encode *LocallyBlocked* sub-formula, we introduce new variables p_i for $i \in \{1 \ldots n\}$, assigning p_i to *true* means component i is part of the subsystem under analysis. This

variables implement the existential 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 form a state in which subsystem ss is blocked. We use $on_i = p_i$ and $on_{i,j} = p_i \vee p_j$.

$$LocallyBlocked \mathrel{\widehat{=}} \Big(\bigwedge_{\substack{i\in\{1\ldots n\}\\ \wedge s\longrightarrow_i}} on_i \Rightarrow \neg st_{i,s}\Big) \wedge \Big(\bigwedge_{\substack{i,j\in\{1\ldots n\}\wedge i\neq j\\ \wedge(s,s')\longrightarrow_{i,j}}} on_{i,j} \Rightarrow \begin{pmatrix} (p_i \wedge \neg st_{i,s}) \\ \vee\, (p_j \wedge \neg st_{j,s'}) \end{pmatrix}\Big)$$

Due to the reuse of formula *Blocked* and *LocallyBlocked*, these encodings can only be applied to triple-disjoint systems. It should be noted, however, that a (locally-)blocked constraint that does not rely on triple-disjointness could be constructed in polynomial time by encoding how components participate on system rules and when they can be triggered.

Sub-formula $Reach_2$ captures the 2-reachability predicate $reach_2$. It examines the state space of pairs of components and disallow pairs of component states that are not pairwise reachable from existing simultaneously.

$$Reach_2 \mathrel{\widehat{=}} \bigwedge_{\substack{i,j\in\{1\ldots n\}\wedge i\neq j\wedge(s,s')\in S_i\times S_j\\ \wedge reachable_{i,j}((s,s'))}} \neg st_{i,s} \vee \neg st_{j,s'}$$

Sub-formulas $Reach_S$, $Reach_S^{DA}$, $Reach'_S$, and $Reach_S^{DA\prime}$ have the same format, given next, but they differ on how they encode the $HBC(i,s)$ constraint.

$$\bigwedge_{i\in\{1\ldots n\}\wedge s\in S'_i} st_{i,s} \Rightarrow HBC(i,s)$$

To encode $HBC(i,s)$, we encode variables clk_k^l and the ordering relationships between them. $Reach_S$ and $Reach_S^{DA}$ encode variables clk_k^l using bit-vectors. Each of these sub-formulas use a different size for their bit-vectors that is given by $\lceil \log_2 |O| \rceil$, where O is the universal set of occurrence, calculated as per Definition 12, for the corresponding predicate[2]. We choose this size of bit-vectors because only $|O|$ distinct clock values are needed to create a model for these sub-formulas. The variables clk_k^l of $Reach_S$ are completely unrelated to the ones of $Reach_S^{DA}$ so we use disjoint sets of boolean variables to encode these two sets of variables. Finally, we encode $<$ as the corresponding operation on bit-vectors. $Reach'_S$ and $Reach_S^{DA\prime}$ are encoded using integer clk_k^l variables and the corresponding $<$ on integers. Again, the clk_k^l variables used by these two sub-formulas are disjoint.

Sub-formulas $Reach_R$, $Reach_R^{DA}$, $Reach_D$, $Reach_D^{DA}$, and $Reach_D^{CA}$ have the same format, given next, but they differ on how they encode the $RC(i,s)$ constraint.

$$\bigwedge_{i\in\{1\ldots n\}\wedge s\in S'_i} st_{i,s} \Rightarrow RC(i,s)$$

To encode $RC(i,s)$, we encode variables N_k and the relations/equations they must respect. $Reach_R$ and $Reach_R^{DA}$ encode variables N_k using bit-vectors of size $|\mathcal{R}|$; again, if a model exists, we should be able to find it with $|\mathcal{R}|$ distinct values for our variables. As the variables N_k of $Reach_S$ are completely unrelated to those of $Reach_S^{DA}$, we use disjoint sets of boolean variables to encode these two sets of variables. Finally, we encode $<$, $=$, and $>$ as the corresponding operation on bit-vectors. $Reach_D$ and $Reach_D^{DA}$, and $Reach_D^{CA}$ are encoded using integer N_k variables, with the restriction they need to be non-negative, and difference equations are trivially encoded using the theory of linear integer arithmetic. Again, the N_k variables used by these three sub-formulas are mutually disjoint.

[2]We can disregard the cases where $|O| = 1$, and later where $|\mathcal{R}| = 1$, as for these boundary cases our techniques are pointless, namely, our predicates are equivalent to *true*.

Broadly speaking, these sub-formulas (namely, the implications they create) ensure that if a component state is assigned to true in a satisfying assignment, the associated reachability constraint is also met. So, any system state satisfying our sub-formulas must pass our reachability tests.

4.3 Practical evaluation

In this section, we evaluate *PairStatic*. We implemented this framework in our DeadlOx tool. It uses FDR4 to generate our SAT and SMT encodings which are then checked by the Glucose 4.0 solver [13] and Z3 solver [39], respectively. FDR4 serves as a library to translate/compile CSP models into supercombinator machines. Also, we use CSP as an input language to describe distributed systems. Note that any other notation could be used if a translation into supercombinator machines is provided. DeadlOx and the models used in this section are available at [4].

Our implementation benefits from the incremental nature of modern SAT and SMT solvers. A formula can be split into several conjuncts, which can be incrementally fed to the solver. This incremental checking is efficient because the solver can use the information obtained from solving one part of the formula to save a lot of effort when re-checking it. So, we check/solve our formulas incrementally. Our implementation starts solving the conjunction of *State*, *Blocked* (or *LocallyBlocked* for local candidates formulas) and $Reach_2$. If no candidates are found, the system is (local-)deadlock free. Otherwise, we add up another reachability test to tighten the state space being analysed, and repeat this solving process. This incremental step continues until the entire formula has been constructed. If a candidate is found for the entire formula, it is reported to the user of our tool. For $PairStatic_R$ and $LocalPairStatic_R$, the other approximations are conjoined in the following order: $Reach_R^{DA}$, $Reach_S^{DA}$, $Reach_R$ and $Reach_S$. For $PairStatic_D$ and $LocalPairStatic_D$, we have the following order: $Reach_D^{DA}$, $Reach_D^{CA}$, $Reach_S^{DA}$, $Reach_D$ and $Reach_S$. These orderings put first the approximations we believe offer a better precision/efficiency compromise.

We extend the input language of FDR4 with the annotations `:[PairStatic]` and `:[PairStatic [smt]]` that should be added to a deadlock free assertion to tell FDR4 to use our $PairStatic_R$ and $PairStatic_D$ encodings, respectively, instead of performing explicit state exploration. For instance, a distributed system described by process `SYSTEM` could be checked using *PairStatic* by the following assertions.

```
assert SYSTEM :[deadlock free [F]] :[PairStatic]
assert SYSTEM :[deadlock free [F]] :[PairStatic [smt]]
```

To check for local-deadlock freedom instead of deadlock freedom, one should change the assertion from `:[deadlock free [F]]` to `:[sublock free [F]]`[3]. By using our *PairStatic* annotations, one can check local-deadlock freedom using our encodings $LocalPairStatic_R$ and $LocalPairStatic_D$. We point out that FDR4 does not have an explicit-exploration engine to check this local-deadlock-freedom assertion. So, for the time being, this type of assertion can only be (approximately) checked by our symbolic approaches.

Our experiment evaluates deadlock and local-deadlock freedom for some triple-disjoint local-deadlock-free systems that cannot be tackled by local analysis alone. Hence, frameworks like *Pair* and SDD (see Section 5) are unable to show (local-)deadlock freedom for all examples discussed in this section. The experiment was conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. In this experiment, we compare DeadlOx's encodings $LocalPairStatic_R$ (PSl), $PairStatic_R$ (PSd),

[3]We extended FDR4 with this new type of assertion in [6].

$LocalPairStatic_D$ (PSl_{smt}) and $PairStatic_D$ (PSd_{smt}) against the Deadlock Checker [57], FDR4's deadlock freedom assertion (FDR) [46], and D-Finder 2 [18]. Deadlock Checker implements the FSDD and CSDD frameworks, FDR4 is a complete method that performs explicit space exploration, and D-Finder 2 is an approximative techniques 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) [47] or compression techniques (FDRc) [75]. While FSDD and CSDD check for a property that is stronger than local-deadlock freedom, FDR4 and D-Finder 2's techniques check for deadlock freedom.

Unsurprisingly, our results suggest that approximate frameworks FSDD, CSDD and *PairStatic* are substantially more scalable than exact methods. Despite being approximate, however, D-Finder 2 performs poorly on all examples considered in this section. It seems that the invariant calculations they carry 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. We split our results into three parts.

Table 1 presents the first part of our results. It presents the running time of DF2pm as it is the best D-Finder 2 technique. It shows the analysis of 9 systems: the butler solution to the dining philosophers where it counts philosophers (*But*), a distributed database (*DDB*), a hexagonal systolic array (*HexSys*) [49], a matrix multiplication system (*Mat*), a ring system implementing a priority-based mutual-exclusion mechanism (*RingP*), a non-fillable ring (*Ring*), Milner's scheduler (*Sched*), a system implementing timestamp-based mutual-exclusion mechanism (*TS*), and a system that implements a majority-vote mutual-exclusion mechanism (*MVote*).

FSDD was designed to show non-fillable systems such as *Ring* deadlock free. CSDD, on the other hand, was conceived to handle systolic-array-like systems such as *HexSys*, *Mat* and *Sched*. *Sched* implements a very basic token mechanism that can also be interpreted as a systolic-array mechanism. The other examples implement mechanisms that are beyond the reach of FSDD and CSDD. We point out that CSDD outperforms *PairStatic* for systolic-array-like systems but we believe that this shortcoming is compensated by *PairStatic*'s improved precision.

In addition to non-fillable and systolic-array-like systems, *PairStatic* can show (local-)deadlock freedom for some counting-based, token-based, priority-based, and timestamp-based systems such as *But*, *DDB*, *RingP* and *TS*, respectively. The versatility of our invariants allows *PairStatic* to capture a variety of behavioural mechanisms. It cannot, though, fully capture the conservation of votes behind the majority-vote mechanism implemented in *MVote*. Hence, it is unable to show (local-)deadlock freedom for this system. Note that the $(Local)PairStatic_D$ encoding can capture the counting mechanism used by *But* to avoid undesired states and the timestamp-based mechanism implemented by *TS*, whereas $(Local)PairStatic_R$ cannot. The reason is that only the use of our component-specific abstraction presented in Section 3.3.2 enables the capture of these mechanisms. The quadratic growth on the number of components as N increases is the reason for the apparent lack of scalability of *PairStatic* for *HexSys* and *Mat*.

Table 2 presents the analysis of some token-based systems. We analyse a message-exchange grid system (*MsgGrid*), a token ring with one token (*Ring2*), a token ring with $N/2$ tokens (RingHf), two simplified versions of these two systems (*Ring2S* and *Ring2SHf*), a train-track system with one train (*Track*) and a train-track system with $N/2$ trains (*TrackHf*);

		Approximate							Exact		
Example	N	FSDD	CSDD	PSl	PSd	PSl_{smt}	PSd_{smt}	DF2pm	FDR	FDRc	FDRp
But	50	-	-	-	-	2.42	1.07	*	*	*	*
	100	-	-	-	-	12.33	4.87	*	*	*	*
	150	-	-	-	-	44.69	17.39	*	*	*	*
	200	*	*	-	-	106.36	32.52	*	*	*	*
DDB	5	-	-	0.16	0.31	0.47	0.36	*	0.51	0.16	0.21
	10	*	*	1.37	1.27	8.03	7.78	*	*	*	*
	15	*	*	11.09	10.53	80.73	79.41	*	*	*	*
	20	*	*	47.26	48.91	*	*	*	*	*	*
HexSys	3	-	0.20	0.21	0.16	0.21	0.16	*	*	0.16	0.36
	5	-	0.28	4.98	3.57	1.77	0.67	*	*	7.33	*
	8	-	0.53	90.40	56.12	50.47	8.13	*	*	*	*
	10	-	0.68	*	160.63	231.88	19.55	*	*	*	*
Mat	5	-	0.18	0.21	0.36	0.21	0.16	*	*	0.21	0.17
	10	-	0.23	5.83	3.87	2.22	0.72	*	*	15.59	0.67
	20	-	0.43	141.76	38.84	55.43	12.24	*	*	*	28.51
	30	-	0.83	*	269.55	*	183.45	*	*	*	*
RingP	5	-	-	0.11	0.22	0.22	0.16	*	0.26	+	0.16
	10	-	-	0.47	0.41	3.77	2.32	*	*	+	*
	15	-	-	2.52	2.32	45.95	24.21	*	*	+	*
	20	*	*	10.59	10.43	224.95	158.51	*	*	+	*
Ring	100	0.18	-	0.16	0.17	0.36	0.26	38.44	*	0.72	*
	200	0.28	-	0.32	0.26	0.92	0.47	*	*	1.57	*
	300	0.38	-	0.47	0.42	1.52	0.61	*	*	2.72	*
	400	0.43	-	0.57	0.51	2.12	0.87	*	*	4.22	*
Sched	100	-	0.18	0.16	0.11	0.42	0.16	5.27	*	0.37	0.41
	200	-	0.28	0.27	0.21	0.72	0.26	10.18	*	0.72	3.82
	300	-	0.38	0.32	0.31	1.32	0.31	18.00	*	1.17	15.74
	400	-	0.43	0.42	0.47	11.69	0.46	47.45	*	1.77	44.68
TS	3	-	-	-	-	0.11	0.11	4.22	0.06	+	0.06
	5	-	-	-	-	0.42	0.36	121.15	0.11	+	0.11
	10	-	-	-	-	7.73	6.78	*	*	+	*
	15	-	-	-	-	69.70	73.05	*	*	+	*
MVote	3	-	-	-	-	-	-	-	0.06	0.11	0.11
	5	-	-	-	-	-	-	-	0.11	0.31	0.16
	7	-	-	-	-	-	-	-	24.46	*	75.55
	10	-	-	-	-	-	-	*	*	*	*

Table 1. Techniques comparison for non-token-based systems. 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. * means that the method took longer than 300 seconds, or an error, such as running out of memory, occurred. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found.

trains can be seen as tokens moving around a network of tracks. Neither FSDD nor CSDD was designed to handle token-based systems so they cannot prove any of these systems deadlock free. $(Local)PairStatic_R$ can capture token mechanisms as long as tokens take a predictable route around the system. Unidirectional token rings such as the ones implemented in *Ring2*, *Ring2Hf*, *Ring2S* and *Ring2SHf* (also *Sched* in Table 1) are predictable enough for $(Local)PairStatic_R$ to capture. On the other hand, the routes around the grid network implemented by *MsgGrid* are too unpredictable to be captured by $(Local)PairStatic_R$ but

		Approximate					Exact		
Example	N	PSl	PSd	PSl_{smt}	PSd_{smt}	DF2	FDR	FDRc	FDRp
MsgGrid	20	-	-	0.21	0.16	15.49	*	0.26	*
	40	-	-	0.31	0.21	*	*	0.77	*
	60	-	-	0.61	0.26	*	*	1.67	*
	80	-	-	0.87	0.36	*	*	3.82	*
Ring2	10	0.31	0.72	1.92	1.47	14.69	0.37	+	0.46
	15	2.02	1.92	24.71	14.99	*	0.46	+	5.27
	20	11.09	10.88	134.43	79.67	*	1.07	+	35.87
	25	41.54	41.49	*	*	*	2.87	+	162.71
Ring2Hf	10	0.31	0.31	1.97	1.42	*	*	+	*
	15	2.02	1.92	26.86	14.24	*	*	+	*
	20	11.19	10.78	137.48	83.42	*	*	+	*
	25	41.79	41.29	*	*	*	*	+	*
Ring2S	50	0.72	1.27	2.92	2.17	217.61	11.60	+	45.03
	100	4.87	4.97	34.23	22.26	*	*	+	*
	150	18.65	19.86	100.81	76.16	*	*	+	*
	200	56.81	55.62	*	261.42	*	*	+	*
Ring2SHf	50	0.72	1.17	2.82	2.62	*	*	+	*
	100	4.92	4.97	24.61	22.31	*	*	+	*
	150	19.30	20.00	112.23	75.06	*	*	+	*
	200	56.38	56.02	*	*	*	*	+	*
Track	100	-	-	-	-	*	1.32	20.05	2.17
	200	-	-	-	-	*	9.18	211.58	23.30
	300	-	-	-	-	*	33.93	*	105.22
	400	-	-	-	-	*	93.79	*	*
TrackHf	100	-	-	-	-	*	*	20.05	*
	200	-	-	-	-	*	*	211.27	*
	300	-	-	-	-	*	*	*	*
	400	-	-	-	-	*	*	*	*

Table 2. Techniques comparison for token-based (deterministic) systems. 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. * means that the method took longer than 300 seconds, or an error, such as running out of memory, occurred. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found.

they can still be captured by $(Local)PairStatic_D$. *Track* and *TrackHf*, however, allow token routes that are too unpredictable even for $(Local)PairStatic_D$; the component-specific abstraction it implements cannot capture these routes. This inability makes *PairStatic* unable to detect that trains (i.e. tokens) cannot be created or destroyed but they can only move around the track. This conservative invariant is essential in proving that these two systems are (local-)deadlock free. We point out that the rapid growth of components as N increases for *Ring2* and *Ring2Hf* makes *PairStatic* less scalable than it is for the other examples.

Table 3 presents the analysis of token networks implementing three communication topologies: fully-connected, grid, bidirectional ring. The name of our examples describe the

		Approximate		Exact		
Example	N	PSl_{smt}	PSd_{smt}	FDR	FDRc	FDRp
TkFully	10	0.16	0.16	0.11	0.16	0.16
	20	0.62	0.52	0.26	0.31	2.22
	30	1.97	1.82	0.82	0.77	22.03
	40	5.32	5.02	2.82	1.37	117.99
TkFullyHf	10	0.16	0.16	45.85	0.16	238.96
	20	0.62	0.57	*	0.31	*
	30	2.12	1.92	*	0.77	*
	40	5.57	5.12	*	1.37	*
TkGrid	40	0.11	0.11	0.12	+	0.21
	60	0.16	0.16	0.11	+	0.46
	80	0.21	0.21	0.16	+	1.02
	100	0.21	0.21	0.21	+	1.97
TkGridHf	40	0.16	0.16	*	*	*
	60	0.21	0.26	*	*	*
	80	0.27	0.32	*	*	*
	100	0.26	0.26	*	*	*
TkRing	40	0.11	0.11	0.11	+	0.11
	60	0.11	0.11	0.12	+	0.16
	80	0.16	0.17	0.16	+	0.22
	100	0.16	0.11	0.16	+	0.31
TkRingHf	40	0.16	0.11	*	0.46	0.31
	60	0.16	0.16	*	0.87	0.56
	80	0.16	0.16	*	1.72	0.97
	100	0.21	0.16	*	3.17	1.52

Table 3. Techniques comparison for token-based non-deterministic networks. 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. * means that the method took longer than 300 seconds, or an error, such as running out of memory, occurred. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found.

topology used. For examples suffixed by *Hf*, components exchange $N/2$ tokens, otherwise they exchange only two. This increase in the number of tokens results in a huge increase in the number of actually reachable system states. These networks implement token routes that can only be captured by our $(Local)PairStatic_D$ encoding. One factor that contributes to the unpredictability of token routes is the multitude of partners a component can pass a token to. Another factor is whether the choice of a communication partner is deterministic or not. While components in the examples in Table 2 deterministically choose a partner to pass the token to. Here, this choice is made non-deterministically. These results show that $(Local)PairStatic_D$, and in particular our component-specific abstraction, can capture token mechanisms that are far from trivial.

This experiment shows that our framework can tackle a relevant class of distributed and concurrent systems. The flexibility of our invariants allows *PairStatic* to capture a variety of interaction mechanisms that are commonly employed by systems to avoid undesired states. The intricate behaviour of our examples makes their analysis far from trivial. Nevertheless,

PairStatic can efficiently show (local-)deadlock freedom for most examples. This experiment suggests that *PairStatic* is marginally less efficient than traditional approximate frameworks. This loss in speed, however, is compensated by a considerable increase in precision. We also point out to the fact that our framework can check local-deadlock freedom in a reasonably scalable way. Even though this property involves ensuring that exponentially many subsystems cannot get irretrievably blocked, the time that our framework needs to verify this property is comparable to the time it takes to check deadlock freedom. These results also demonstrate a shortcoming of our framework. For some systems, (local-)deadlock freedom depends on the fact that tokens are conserved, namely, tokens can be exchanged between components but they cannot be created or destroyed. To capture this invariant, our framework must, to some extent, correctly identify the routes tokens take around the system. In some cases, however, this route is too unpredictable for *PairStatic* to capture. In these cases, our framework is unable to prove deadlock or local-deadlock freedom. In terms of efficiency, we point out that in some cases the size of components grows substantially as N grows. This happens for instance for examples *DDB*, *RingP*, *TS*, *Ring2*, and Ring2H. For such cases, our framework is not so scalable.

5 RELATED WORK

Many approximate techniques have harnessed local analysis to verify systems in several contexts and using different types of concurrency [7, 8, 10, 11, 52, 57, 62]. They are all built, to some extent, around the *fundamental principle*: under reasonable assumptions about the system, a cycle of ungranted requests is a necessary condition for a deadlock. So, they prove systems are free of such cycles and consequently deadlock free. These methods tend to be very efficient, they normally rely on some polynomially checkable condition, but the characterisation of deadlock candidates as cycles of dependencies can be imprecise in many ways [2]. Absence of such cycles is, in fact, a property that is stronger than local-deadlock freedom. Moreover, techniques based on pure local analysis cannot capture invariants emerging from the global behaviour of the system.

To circumvent this issue, many techniques to perform efficient global analysis were proposed. Martin's CSDD and FSDD as discussed in Section 4.1 are examples of such framework. Data-flow analysis inspired our work on synchronisation analysis. It is an approach often used to approximate the behaviour of sequential and concurrent systems [20, 22, 28, 41, 70].

The work in [28] proposes two algorithms that use data-flow analysis to approximate reachability. The first one analyses the interaction structure of the system and conservatively marks which component states are reachable and which are not. The second algorithm improves on the precision of the first one by adding some extra information about the "history" of components. This information helps eliminate some interactions between components that were conservatively assumed by the first algorithm. These algorithms only work for systems with unreachable individual component states. Well-designed systems, however, are expected to reach every component state at some point.

The work in [41] introduces an approximate framework that uses data-flow analysis to analyse concurrent programs. It extracts control-flow graphs annotated with events from Ada concurrent programs. Based on those graphs, it builds a trace-flow graph (TFG): a conservative and compact representation for all the traces the concurrent program can engage on. So, it uses data-flow analysis to check whether the TFG respects a given property, described as a finite-state automaton. A property is tested by analysing whether the terminating traces of the TFG are accepted by the automaton. We point out that the TFG representation only enforces some weak interaction restrictions for the system.

Hence, it represents a fairly loose approximation for the system's behaviour. To cope with that, this work allows the user to manually add some constraints that help tighten this approximation. It advocates for an interactive approach where, based on a counter-example, the user identifies whether it is spurious and, if so, designs some constraint to eliminate it.

To some extent, both these approaches use abstract-interpretation ingredients to approximate reachability. The way they use these ingredients and how they test for reachability, however, is significantly different from what we do.

Other frameworks rely on specific techniques to find global system invariants which capture compact abstractions for the behaviour of concurrent and distributed systems [9, 19, 53]. D-Finder 2, for instance, is a deadlock-checking tool that handles component-based systems described in the BIP notation [17, 18]. This framework can handle systems with infinite-state components as it employs a mechanism to find finite-state abstractions for those. Given finite-state abstractions for components, it derives an *interaction invariant* which constitutes an over-approximation for the set of reachable states. It does so by solving a set of implications that computes sets X of component states such that all reachable system state have a component state in X; these sets can be understood as traps in the Petri-net setting [60]. It uses three methods to compute these sets: an enumerative method, a boolean-constraint-based one and a fixed-point-based one. All of these methods might compute a number of sets that is exponentially large on the size of the system.

In this work, we have propose and studied an approximate framework to check deadlock and local-deadlock freedom. We point out, however, that synchronisation analysis, and the approximations they capture, can be easily fitted into a framework to check more general properties. In [5], we propose a framework that relies on (some of) the reachability approximations presented here to check *static properties*, namely, properties that can be naturally formulated as "a system cannot reach a state in which it can perform a given combination of events". Many other frameworks rely instead on necessary conditions that are inherent to the properties they check [42, 63, 64]. For instance, the cycle-of-dependencies condition checked by SDD is inherent to deadlock analysis, whereas frameworks in [42, 63, 64] use some specific necessary conditions to show livelock freedom and determinism. The use of these property-specific conditions makes these frameworks difficult to adapt to verifying other properties.

6 CONCLUSION

This paper introduces the concept of *synchronisation analysis* to capture global invariants and approximate reachability. It uses a component-synchronisation-analysis framework to calculate invariants on how components participate on (global) system synchronisations/interactions and on a notion of consistency between these invariants to establish whether components can effectively communicate to reach some system state. Our CSA uses abstract-interpretation elements that are normally part of data-flow-analysis frameworks but in a rather different way. So, as far as the authors of this paper are aware, our idea of synchronisation analysis and the techniques we implement are new.

We introduce three synchronisation-analysis techniques: the first technique tries to show that a system state is unreachable by demonstrating that components cannot agree on the order they participate in system rules, whereas the second and third techniques try to establish that a system state is unreachable by demonstrating components cannot agree on the number of times they participate on system rules. These techniques are imprecise in the sense that they either establish that a system state is unreachable, or they are unable to do so and we conservatively assume the system state is reachable. These techniques can,

in particular, capture invariants that are notably used to prove properties of non-fillable, systolic-array-like and token-based systems.

We combine these approximations with 2-reachability presented in [2] to create the *PairStatic* framework. These new approximations allow this framework to capture global invariants of the system as long as they can be represented by a combination of our component-state invariants. Hence, it can prove deadlock (and local-deadlock) free systems that are beyond the capabilities of *Pair*. We present some experimental evidence that suggests *PairStatic* is able to prove (local-)deadlock freedom for an interesting class of distributed and concurrent system and it does so in a scalable way. Many commonly employed interaction paradigms can be efficiently captured by our framework. It also suggests that the approximations derived from the sort of synchronisation analysis we propose can be efficiently checked by SAT/SMT solving. So, it could be used as a preliminary step in deadlock-freedom checking. If it fails to prove deadlock freedom, then a precise method should be used. Given the apparent complexity of checking local-deadlock freedom, our framework might be the only practical way to check this property.

Our synchronisation-analysis techniques were inspired by Martins's CSDD and FSDD, which were in turn inspired by proof rules from [74]. We have, however, removed some of FSDD and CSDD's limitations. In particular, we propose reachability approximations that are completely independent of the safety property that is being checked, while both the CSDD and FSDD focus on a condition that is inherently linked to deadlock analysis. Furthermore, we point out that a CSA framework could be used to calculate the last-action and number-of-cycles invariants used by FSDD and CSDD. So, our approach could implement the same analysis they propose which could possibly speed up our analysis of systolic-array-like systems.

Our implementation is restricted to pairwise-communicating (i.e. triple-disjoint) systems so we can re-use *Pair*'s efficient strategy to encode the *blocked* predicate. Our reachability tests and their encodings, however, can be applied to systems with multiway communication. Moreover, the ideas in this paper should transfer easily to any formalism where systems are described by interacting LTSs. DeadlOx uses FDR4 to obtain supercombinator machines from systems described using CSP, but a tool analogous to DeadlOx could be created for other notations by replacing its use of FDR4 to generate such machines.

We plan to extend the work presented in this paper in two ways. Firstly, we plan to propose methods to reduce the size of our encodings. In some cases, we could simplify our formula by abstracting away or merging together some invariants. The way that CSDD counts number of cycles instead of number of rules application is an example of such abstractions. Secondly, we intend to investigate what other synchronisation-analysis frameworks can be created and which sort of invariants can we capture with these new frameworks. Can we create more complex frameworks leading to stronger invariants? How this affects the efficiency of SAT/SMT solving? For instance, we plan to create and investigate a framework that analyses the (possibly many) last actions that lead a component to its state.

ACKNOWLEDGMENTS

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 number EP/N022777, and Innovate UK and the Aerospace Technology Institute via the SECT-AIR project under agreement number 113099.

REFERENCES

[1] Antonino, P. *Verifying concurrent systems by approximations*. DPhil thesis, University of Oxford, Under submission. Available at: www.cs.ox.ac.uk/people/pedro.antonino/the.pdf.

[2] Antonino, P., Gibson-Robinson, T., and Roscoe, A. Efficient deadlock-freedom checking using local analysis and SAT solving. In *IFM* (2016), no. 9681 in LNCS, Springer, pp. 345–360.

[3] Antonino, P., Gibson-Robinson, T., and Roscoe, A. Tighter reachability criteria for deadlock freedom analysis. In *FM* (2016), no. 9995 in LNCS, Springer.

[4] Antonino, P., Gibson-Robinson, T., and Roscoe, A. Experiment package, 2018. Available at: `www.cs.ox.ac.uk/people/pedro.antonino/thepkg.zip`.

[5] Antonino, P., Gibson-Robinson, T., and Roscoe, A. W. Checking static properties using conservative SAT approximations for reachability. In *SBMF* (2017), pp. 233–250.

[6] Antonino, P., Gibson-Robinson, T., and Roscoe, A. W. Efficient verification of concurrent systems using local-analysis-based approximation and sat solving. Manuscript submitted to Formal Aspects of Computing. Available at: www.cs.ox.ac.uk/people/pedro.antonino/local.pdf, 2018.

[7] Antonino, P., Oliveira, M. M., Sampaio, A., Kristensen, K., and Bryans, J. Leadership election: An industrial SoS application of compositional deadlock verification. In *NFM* (2014), vol. 8430 of *LNCS*, pp. 31–45.

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

[9] Apt, K. R., Francez, N., and De Roever, W. P. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS) 2*, 3 (1980), 359–385.

[10] Attie, P. C., Bensalem, S., Bozga, M., Jaber, M., Sifakis, J., and Zaraket, F. A. An Abstract Framework for Deadlock Prevention in BIP. In *FORTE*, no. 7892 in LNCS. Springer, 2013, pp. 161–177.

[11] Attie, P. C., Bensalem, S., Bozga, M., Jaber, M., Sifakis, J., and Zaraket, F. A. Global and local deadlock freedom in BIP. *ACM Trans. Softw. Eng. Methodol. 26*, 3 (2018), 9:1–9:48.

[12] Attie, P. C., and Chockler, H. Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In *VMCAI* (2005), Springer, pp. 465–481.

[13] Audemard, G., and Simon, L. Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI'09, pp. 399–404.

[14] Baier, C., and Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[15] Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T. H., and Sifakis, J. Rigorous component-based system design using the bip framework. *IEEE Software 28*, 3 (2011), 41–48.

[16] Bellman, R. On a routing problem. *Quarterly of Applied Mathematics 16*, 1 (1958), 87–90.

[17] Bensalem, S., Bozga, M., Legay, A., Nguyen, T., Sifakis, J., and Yan, R. Component-based verification using incremental design and invariants. *Software and System Modeling 15*, 2 (2016), 427–451.

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

[19] Bensalem, S., and Lakhnech, Y. Automatic Generation of Invariants. *Form. Methods Syst. Des. 15*, 1 (July 1999), 75–92.

[20] Beyer, D., and Keremoglu, M. E. Cpachecker: A tool for configurable software verification. In *CAV 2011* (Berlin, Heidelberg, 2011), G. Gopalakrishnan and S. Qadeer, Eds., Springer Berlin Heidelberg, pp. 184–190.

[21] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems* (1999), 193–207.

[22] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *PLDI 2003* (2003), PLDI '03, ACM, pp. 196–207.

[23] Brookes, S. D., and Roscoe, A. W. Deadlock analysis in networks of communicating processes. *Distributed Computing* **4** (1991), 209–230.

[24] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L.-J. Symbolic model checking: 1020 states and beyond. *Information and computation 98*, 2 (1992), 142–170.

[25] Canfora, G., and Di Penta, M. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, 2006, pp. 78–105.

[26] Cavalcanti, A. Formal methods for robotics: Robochart, robosim, and more. In *Formal Methods: Foundations and Applications* (Cham, 2017), S. Cavalheiro and J. Fiadeiro, Eds., Springer International

Publishing, pp. 3–6.
[27] Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., and Sinha, N. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing 17*, 4 (2005), 461–483.
[28] Cheung, S. C., and Kramer, J. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering 20*, 8 (Aug 1994), 579–593.
[29] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *Computer aided verification* (2000), Springer, pp. 154–169.
[30] Clarke, E. M., Grumberg, O., and Peled, D. *Model checking*. MIT press, 1999.
[31] Clarke, E. M., and Wing, J. M. Formal methods: State of the art and future directions. *ACM Comput. Surv. 28*, 4 (1996), 626–643.
[32] Coleman, J. W., Malmos, A. K., Larsen, P. G., Peleska, J., Hains, R., Andrews, Z., Payne, R., Foster, S., Miyazawa, A., Bertolini, C., and Didierk, A. Compass tool vision for a system of systems collaborative development environment. In *2012 7th International Conference on System of Systems Engineering (SoSE)* (2012), pp. 451–456.
[33] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.
[34] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1977), POPL '77, pp. 238–252.
[35] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1979), POPL '79, pp. 269–282.
[36] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varro, D. Viatra - visual automated transformations for formal verification and validation of uml models. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,* (2002), pp. 267–270.
[37] Dathi, N. *Deadlock and Deadlock Freedom*. PhD thesis, University of Oxford, 1989.
[38] de Alfaro, L., and Henzinger, T. A. Interface theories for component-based design. In *Embedded Software* (Berlin, Heidelberg, 2001), T. A. Henzinger and C. M. Kirsch, Eds., Springer Berlin Heidelberg, pp. 148–165.
[39] de Moura, L. M., and Bjørner, N. Z3: An efficient smt solver. In *TACAS* (2008), pp. 337–340.
[40] D'silva, V., Kroening, D., and Weissenbacher, G. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27*, 7 (2008), 1165–1178.
[41] Dwyer, M. B., Clarke, L. A., Cobleigh, J. M., and Naumovich, G. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol. 13*, 4 (Oct. 2004), 359–430.
[42] Filho, M. S. C., Oliveira, M. V. M., Sampaio, A., and Cavalcanti, A. Local livelock analysis of component-based models. In *ICFEM* (2016), pp. 279–295.
[43] Fitzgerald, J., Bryans, J., and Payne, R. A formal model-based approach to engineering systems-of-systems. In *Collaborative Networks in the Internet of Services* (Berlin, Heidelberg, 2012), L. M. Camarinha-Matos, L. Xu, and H. Afsarmanesh, Eds., Springer Berlin Heidelberg, pp. 53–62.
[44] Ford, D. R., and Fulkerson, D. R. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.
[45] France, R., and Rumpe, B. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering* (2007), FOSE '07, pp. 37–54.
[46] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., and Roscoe, A. FDR3 — A Modern Refinement Checker for CSP. In *TACAS* (2014), vol. 8413 of *LNCS*, pp. 187–201.
[47] Gibson-Robinson, T., Hansen, H., Roscoe, A., and Wang, X. Practical partial order reduction for CSP. In *NFM* (2015), vol. 9058 of *LNCS*, Springer, pp. 188–203.
[48] Godefroid, P., and Wolper, P. Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD* **2**, **2** (1993), 149–164.
[49] Gruner, S., and Steyn, T. J. Deadlock-freeness of hexagonal systolic arrays. *Inf. Process. Lett. 110*, 14-15 (July 2010), 539–543.
[50] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[51] Kroening, D., and Strichman, O. *Decision Procedures: An Algorithmic Point of View*, 1 ed. Springer Publishing Company, Incorporated, 2008.
[52] Lambertz, C., and Majster-Cederbaum, M. Analyzing Component-Based Systems on the Basis of

Architectural Constraints. In *FSEN*. Springer, Apr. 2011, pp. 64–79.
[53] Lamport, L. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, 2 (1977), 125–143.
[54] Latella, D., Majzik, I., and Massink, M. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Aspects of Computing 11*, 6 (Dec 1999), 637–664.
[55] Lilius, J., and Paltor, I. P. vuml: a tool for verifying uml models. In *14th IEEE International Conference on Automated Software Engineering* (1999), pp. 255–258.
[56] Lima, L., Miyazawa, A., Cavalcanti, A., Cornélio, M., Iyoda, J., Sampaio, A., Hains, R., Larkham, A., and Lewis, V. An integrated semantics for reasoning about sysml design models using refinement. *Software and System Modeling 16*, 3 (2017), 875–902.
[57] Martin, J., and Jassim, S. An efficient technique for deadlock analysis of large scale process networks. In *FME '97* (1997), pp. 418–441.
[58] Martin, J. M. R. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.
[59] Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., and Timmis, J. Automatic property checking of robotic applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 3869–3876.
[60] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (Apr 1989), 541–580.
[61] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
[62] Oliveira, M. V. M., Antonino, P., Ramos, R., Sampaio, A., Mota, A., and Roscoe, A. W. Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing* (2016), 1–68.
[63] Otoni, R., Cavalcanti, A., and Sampaio, A. Local analysis of determinism for CSP. In *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings* (2017), pp. 107–124.
[64] Ouaknine, J., Palikareva, H., Roscoe, A. W., and Worrell, J. A static analysis framework for livelock freedom in CSP. *LMCS* **9**, **3** (2013).
[65] Paige, R., and Tarjan, R. E. Three partition refinement algorithms. *SIAM Journal on Computing 16*, 6 (1987), 973–989.
[66] Palikareva, H., Ouaknine, J., and Roscoe, A. SAT-solving in CSP trace refinement. *Science of Computer Programming 77*, 10 (2012), 1178 – 1197.
[67] Peled, D. All from one, one for all: on model checking using representatives. In *Computer Aided Verification* (1993), Springer, pp. 409–423.
[68] Plotkin, G. A structural approach to operational semantics. Tech. rep., DAIMI FN-19, Computer Science Dept, Aarhus University, 1981.
[69] Ramos, R. T. *Systematic Development of Trustworthy Component-based Systems*. PhD thesis, Universidade Federal de Pernambuco, 2011.
[70] Reif, J. H., and Smolka, S. A. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming 19*, 1 (Feb 1990), 1–30.
[71] Roscoe, A. *Understanding Concurrent Systems*. Springer, 2010.
[72] Roscoe, A. W. Routing messages through networks: an exercise in deadlock avoidance. In *Programming of Transputer Based Machines: Proceedings of 7th occam User Group Technical Meeting* (Amsterdam, 1987), M. et al., Ed., IOS B.V.
[73] Roscoe, A. W. *The theory and practice of concurrency*. Prentice Hall, 1998.
[74] Roscoe, A. W., and Dathi, N. The pursuit of deadlock freedom. *Inf. Comput.* **75**, **3** (1987), 289–327.
[75] Roscoe, A. W., Gardiner, P. H. B., Goldsmith, M., Hulance, J. R., Jackson, D. M., and Scattergood, J. B. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS* (1995), pp. 133–152.
[76] Scholten, C. S., and Dijkstra, E. W. *A Class of Simple Communication Patterns*. Springer New York, New York, NY, 1982, pp. 334–337.
[77] Selic, B. The pragmatics of model-driven development. *IEEE Software 20*, 5 (2003), 19–25.
[78] Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics 5*, 2 (1955), 285–309.
[79] Valmari, A. A stubborn attack on state explosion. *Formal Methods in System Design 1*, 4 (1992),

297–322.

[80] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. S. Formal methods: Practice and experience. *ACM Comput. Surv. 41*, 4 (2009).

[81] Yeh, W. J., and Young, M. Compositional reachability analysis using process algebra. In *Proceedings of the symposium on Testing, analysis, and verification* (1991), ACM, pp. 49–59.

Received May 2018