Verifying concurrent systems by approximation

Pedro Ribeiro Gonçalves Antonino

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2018

This thesis is dedicated to Totia

Acknowledgements

Firstly, I would like to thank my supervisor Bill Roscoe for his constant support and encouragement. He has been very kind and patient with me and an inexhaustible source of advice and knowledge during my studies. I also thank Tom Gibson-Robinson for his attention and help during all my (mostly impromptu) visits to ask questions about the ins and outs of FDR.

I am grateful to Gavin Löwe and Markus Roggenbach for providing very insightful comments and suggestions that helped to improve this thesis.

I must also thank Augusto, my MSc supervisor, friend and mentor. He introduced me to the area of formal verification and has always been an invaluable and unwavering source of advice and support.

I would like to thank my office mates for providing a comfortable research environment. Also, OUGS for organising some much-needed football games on a weekly basis. Also, I would like to thank my Brazilian friends for sharing some de-stressing surfing sessions during my holidays in Brazil and my family for being a constant source of love and care.

I owe my parents a huge thanks for always inspiring me, since a very young age, to fulfil all my dreams. They have relentlessly encouraged and supported me in my pursuit of further studies.

Most importantly, I cannot thank my Talitinha enough for always being there for me. She has provided me with a vital daily dose of love, encouragement, support and laughter. She was an integral part in keeping my sanity during this whole process.

Lastly, I thank the CAPES foundation for financially supporting me.

Verifying concurrent systems by approximation

Pedro Ribeiro Gonçalves Antonino

Wolfson College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2018

Approximate verification frameworks are an approach to combat the well-known state-space explosion problem. For properties formulated as "no bad state can be reached", an approximate framework can be simply obtained by replacing exact reachability by some over-approximation. These frameworks look for a bad state in this over-approximation. If no bad states are found, the system satisfies the original property; the framework is sound. If a bad state is found, we have an inconclusive result: this bad state might be reachable or not. This permitted incompleteness is a cornerstone of such methods and it is a means to obtain efficiency.

In this thesis, we propose three techniques to approximate reachability. The first analyses small subsystems to capture locally provable relationships between component states. The second and third complement the first by capturing some global system invariants. While the second combines component invariants to estimate whether components can cooperate to reach a system state, the third technique detects token mechanisms and deduces token invariants that approximate reachability.

Moreover, we show how our approximations can give rise to useful frameworks that check deadlock freedom, static properties – a class of state-based properties we propose – and CSP's traces and failures refinement expressions. Our evaluation of these frameworks seems to suggest that they can efficiently prove a large spectrum of properties for a variety of systems.

Despite tackling *NP*-hard problems, our frameworks can be efficiently powered by SAT/SMT solvers. The unsuccessful attempts at using symbolic exploration to precisely check concurrent systems, combined with the success of our frameworks, suggest that the approximations we study are a key factor in harnessing the verification power of these solvers.

Contents

1 Introduction **1**

1.1 Related Work 6

1.2 Outline 10

2 Background **13**

2.1 CSP 13

2.1.1 Syntax 14

2.1.2 SOS-style operational semantics 15

2.1.3 Denotational Semantics 16

2.1.3.1 Traces 17

2.1.3.2 Stable failures 18

2.1.3.3 Refinement relations 19

2.2 FDR4 21

2.2.1 Combinator-style operational semantics 21

2.2.2 Property and refinement checking 25

2.2.3 Taming the state-space explosion problem 35

2.2.3.1 Partial order reduction 35

2.2.3.2 Compression 36

2.3 SAT solving 37

2.3.1 The DPLL framework 38

2.3.2 Decision heuristic 40

2.3.3 Conflict-driven backtracking 42

2.3.4 Encoding compactness and quality 44

2.4 SMT solving 45

2.4.1 Linear integer arithmetic 46

2.4.2 The DPLL(T) framework 47

2.5 Summary 50

3 Local analysis **53**
3.1 Introduction 53
3.2 Subsystem reachability 54
3.2.1 k-reachability 56
3.3 Pair: 2-reachability for deadlock freedom 59
3.3.1 Precision of Pair 60
3.3.2 Complexity of Pair 66
3.3.3 Pair-candidate detection via SAT solving 70
3.3.4 Practical evaluation 72
3.4 PairPicking: Pair meets user's picks 76
3.4.1 Examples and practical evaluation 78
3.5 Conclusion 80

4 Global analysis using synchronisation analysis **85**
4.1 Introduction 85
4.2 Approximate reachability via synchronisation analysis 86
4.2.1 Component-synchronisation analysis 87
4.2.2 Synchronisation analysis 90
4.2.2.1 Ordering of rule occurrences consistency 92
4.2.2.2 Number of rules occurrences consistency 98
4.2.3 Rule abstraction 105
4.2.3.1 Data abstraction 105
4.2.3.2 Component-specific abstractions 108
4.2.4 Discussion 110
4.3 PairStatic: Pair plus synchronisation analysis 111
4.3.1 Precision and complexity of PairStatic 112
4.3.2 PairStatic-candidate detection via SAT/SMT solving 118
4.3.3 Practical evaluation 120
4.4 Conclusion 126

5 Global analysis via token structures and invariants **129**
5.1 Introduction 129
5.2 Approximate reachability via token structures and invariants 130
5.2.1 Binary conservative technique 132
5.2.2 Binary existential technique 137
5.2.3 Counter-example-guided conservative technique 141
5.2.4 Counter-example-guided existential technique 143

5.2.5 Discussion 146
5.3 PairToken: Pair meets token invariants 148
5.3.1 Precision and complexity of PairToken 148
5.3.2 PairToken-candidate detection via SAT/SMT solving 156
5.3.3 Practical evaluation 158
5.4 Conclusion 165

6 Approximate verification of static properties 169
6.1 Introduction 169
6.2 Capturing static properties 171
6.2.1 Static properties 171
6.2.2 Checking static properties 173
6.2.3 Capturing some other static properties 173
6.3 Approximate verification of static properties 176
6.3.1 Precision and complexity of StaticProperty 177
6.3.2 Static-property checking via SAT/SMT solving 182
6.3.3 Practical evaluation 184
6.3.3.1 Checking deadlock and local-deadlock freedom 185
6.3.3.2 Checking other properties 189
6.4 Conclusion 191

7 Approximate refinement checking 193
7.1 Introduction 193
7.2 Approximate refinement checking 194
7.2.1 Watchdog-based refinement checking 194
7.2.2 Approximate framework 198
7.2.3 Precision and complexity of RefinementChecking 200
7.2.4 Refinement checking via SAT/SMT solving 212
7.2.5 Practical evaluation 214
7.2.5.1 Traces refinement checking 214
7.2.5.2 Stable-failures refinement checking 217
7.3 Conclusion 219

8 Conclusion 221
8.1 Limitations and future work 229

Bibliography 232

List of Figures

2.1 SOS rules 17

2.2 LTSs of components C_0 and C_1 above that of SC_2. 18

2.3 LTS sketch of component CC for $\Sigma_I = \{e_1, \ldots, e_m\}$. 29

2.4 LTS sketch of component S_i with $\{e \mid s_i \xrightarrow{e}\} = \{e_1 \ldots e_{m_i}\}$ 30

2.5 LTS of component C_i. 31

2.6 Possible partial DPLL-framework run for Running Example 2.3. . . . 41

2.7 Implication graph at Step 6 of execution in Figure 2.6. 43

3.1 LTSs of components L_1, L_2 and L_3, respectively. 60

3.2 The relationship between deadlock-freedom-checking frameworks. . . 64

3.3 LTS of philosopher i. 65

3.4 LTS of fork i and transitions of the butler process. 66

3.5 LTSs F_i and X_i respectively. 68

4.1 Example of components L_1 and L_2. 87

4.2 LTS of component L_i where $\oplus$ represents addition modulo 3. 92

4.3 LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively. 98

4.4 LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively. 103

4.5 LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively. 106

4.6 LTSs of components L_0, L_1, and L_2, respectively. 109

4.7 The relationship between deadlock-freedom-checking frameworks. . . 113

4.8 LTS of component L_i where $\oplus$ represents addition modulo 3. 114

4.9 Sketch of LTS for components $L_{i,j}$. 116

4.10 Sketch of LTS for components L_i. 117

5.1 LTSs of components L_0, L_1, and L_2, respectively. 137

5.2 LTSs of components L_0, L_1, and L_2, respectively. 141

5.3 LTSs of components L_0, L_1 and L_2, respectively. 146

5.4 LTSs of butler B and of fork F_i, respectively. 151

5.5 LTSs of philosopher P_i. 151

5.6 LTS of component L_i where $\oplus$ represents addition modulo 3. 153
5.7 LTSs of components L_0, L_1, and L_2, respectively. 154
5.8 The relationship between deadlock-freedom-checking frameworks. . . 155

6.1 LTSs of Philosophers 0 and 1, respectively. 178
6.2 LTS of Fork i. 178
6.3 LTS of component L_i where $\oplus$ represents addition modulo 3. 179
6.4 LTSs of components L_0, L_1, and L_2, respectively. 180

7.1 LTSs L_0 and L_i for $i \in \{1, 2\}$, respectively. 201
7.2 GLTS L_{sp} with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$. 201
7.3 LTSs L_i for $i \in \{0, 1, 2\}$ and L_3, respectively. 202
7.4 GLTS L_{sp} with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$. 203
7.5 LTSs of Philosophers 0 and i for $i \in \{1, 2, 3\}$, where $\oplus$ represents subtraction modulo 4, respectively. 204
7.6 LTS of Fork i, where $\ominus$ represents subtraction modulo 4, and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively. . . 204
7.7 LTSs of components L_0 and L_1, respectively. 205
7.8 LTS of component L_2 and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively. 205
7.9 LTS of component L_i for $i \in \{0, 1, 2\}$, where $\oplus$ represents addition modulo 3, and GLTS L_{sp}, respectively. 207
7.10 LTSs F_i and X_i respectively. 210
7.11 LTS CC and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively. 210

Chapter 1

Introduction

Ensuring the correctness of computational systems is becoming increasingly important and complex. Computational systems are ubiquitous in present days. Programs are running on a variety of devices ranging from toasters to airplanes, and they carry out some core functions of these systems [Eng, WLBF09]. A malfunction of such systems normally leads to substantial financial losses but, in some critical contexts, it might even lead to the loss of human lives [Wir, Hor]. To make matters worse, the complexity of systems is increasing. Modern systems usually combine complex components that cooperate to perform some intricate tasks. Therefore, to make sure that systems correctly perform their core functionalities, we need verification techniques and tools that tame their ever increasing complexity. For instance, we want to make sure that the program/controller managing the altitude of the plane is always responsive to commands issued by the pilot. Or, that when the toaster is turned on, its internal heater will eventually turn off, otherwise the bread will burn or, even worse, the entire toaster might catch on fire.

A particular source of complexity arises from the need for components to cooperate. In most cases, components can interact in many different ways. So, to ensure they cooperate effectively, verification techniques need to account for all these possibilities. Effective cooperation often entails *deadlock freedom*: ensuring the system cannot reach a point in which all components are stuck. Testing, a particularly useful technique in improving the quality of systems in the sequential world, for instance, 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 [CDP06].

Formal verification is an alternative to testing that tends to be more effective in ensuring the quality of concurrent and distributed systems [WLBF09, CW96]. While testing ensures that the system behaves correctly for some particular cases, formal verification relies on mathematical foundations to ensure it always behaves

appropriately. Generally, testing involves experimenting on the physical system at hand, while verification techniques analyse mathematical models of systems. There exists a broad spectrum of verification techniques ranging from interactive to fully-automated ones. *Interactive techniques*, implemented in *interactive theorem provers*, offer a body of mathematical theories that allows one to reason about systems and prove them correct [NPW02, HKPM97, COR+95]. These techniques can be much more effective in proving that systems are correct than fully-automated ones. Nevertheless, this effectiveness depends substantially on the ingenuity of their user. Despite offering some degree of automation, these techniques largely rely on the user's manual input and ability to guide the system's verification. On the other hand, *fully-automated techniques*, implemented, for instance, in *model checkers*, are simple-to-use push-button frameworks [DKW08, BK08, CGP99]. Given a description of the system and its specification, these techniques automatically yield whether the system meets its specification or not. In the negative case, they also provide a counter-example, namely, a possible behaviour of the system that violates its specification. Counter-examples provide a very useful piece of information that helps refine the system's design, i.e. correct some bugs, so it can meet its intended specification. As these techniques are fully automatic, simpler to use, and easier to learn, they are normally preferred over interactive ones. The automatic techniques, however, are prone to inefficiency; they are considerably hindered by the vast number of ways in which components can interact. We point out that, typically, verification is carried out for some specific properties of interest, such as the ones we described for airplanes and toasters in the first paragraph, as opposed to complete specifications describing how the entire system should behave.

Model checkers rely on state-space exploration to ensure that a system behaves as intended. These tools traditionally represent a system by its components, each of which is described as a finite-state machine, and rules that describe how components interact. In this work, we use the *supercombinator machine* notation to describe systems [Ros10]. The state space of the system is given by reachable combinations of component states, one per component. Model checkers construct and explore this product space to make sure the system cannot reach a *bad* state. The vast number of ways in which components can cooperate gives rise to *the state-space explosion problem*: the state space of distributed and concurrent systems grows exponentially on the number of components. This rapid growth means that analysing systems is often infeasible even for rather small number of components. This problem is arguably the most intrinsic and debilitating issue in the area of fully-automated verification and so

it has fuelled research for quite some time; finding techniques to tackle the state-space explosion has been an active area of research since model checkers were invented.

A number of techniques have been proposed to tackle the state-space explosion problem such as partial-order reductions [GW93, GRHRW15, Val92, Pel93], compression techniques (or, compositional reachability analysis) [RGG+95, YY91], symbolic state-space representation [POR12, BCCZ99, BCM+92] and counter-example-guided abstraction refinement (CEGAR) [CGJ+00, CCO+05]. They employ different mechanisms to reduce the state space to be explored. For instance, partial-order reduction identifies some similar ways in which components can cooperate. Hence, to check whether a property holds, fewer possibilities of cooperation need to be explored. CEGAR, on the other hand, systematically refines an abstract version of the system until it converges to a representation that is faithful enough to check the desired property. All these techniques have in common their quest for a precise reduction/abstraction of the original state space. Despite being able to tame the state-space explosion problem in many cases, the exact nature of these reductions mean they are bound to be as inefficient as simple explicit state-space exploration in some other cases. In fact, for many of the systems that we have analysed in this thesis, these techniques are even less efficient than simple explicit state-space exploration because the state-space reduction does not compensate the time taken to carry it out.

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

A proxy property can be simply created by means of a *reachability approximation*. Many properties are naturally formulated as (or, can be simply translated into) "no bad state can be reached", where a bad state accounts for some erroneous behaviour. For such properties, replacing exact reachability by some over-approximation creates a proxy property. For instance, deadlock freedom is formulated as "no deadlocked state is in the set of reachable states", and its approximate/proxy counterpart as "no deadlocked state lies within the over-approximation". If this approximation tightly captures the actual state space of a system, it can give rise to a reasonably accurate approximative framework. Obviously, we expect the use of this approximation to speed up the verification process.

In many verification formalisms, reachability is somewhat built in (i.e. a somewhat implicit aspect of) the satisfaction relation they propose. For instance, some frameworks rely on a refinement relation that checks a notion of compatibility between the *reachable* states of specification and implementation. To a certain extent, these formalisms tend to focus on the notion of compatibility between states, whereas reachability is just taken for granted. In our work, however, the reachability analysis is explicit; we investigate verification problems that are formulated as a sort of reachability query. We propose a number of techniques to approximate reachability and we analyse the verification frameworks they give rise to, in particular, their precision and scalability. We begin by using our approximation to check deadlock-freedom, which is perhaps the most fundamental property expected of distributed systems, and we later show how we can extend their use to further properties.

The first reachability approximation we propose is based on *local analysis*. Verification frameworks analyse the global behaviour of the system to yield whether a property holds or not. In many cases, however, a property can be established based on the analysis of small parts of the system. Chapter 3 introduces a reachability approximation that is constructed based on the analysis of pairs of components. We use this approximation to check deadlock freedom in the *Pair* framework. This framework is reasonably scalable and precise. For instance, it can show deadlock freedom for some well-behaved resource-allocation and client-server systems. Nevertheless, Pair cannot show deadlock freedom when it depends on some global invariant of the system. This shortcoming is common to all strategies using pure local analysis. This work has been partially presented in [AGRR16a].

We improve on Pair by proposing a few techniques to approximate reachability based on the system's global behaviour. In Chapter 4, we propose techniques that are based on *synchronisation analysis*. We use a data-flow-analysis-inspired framework

to summarise the behaviour of components and use this summary to detect whether they can consistently cooperate to reach a state. While one approach tries to check whether components can consistently agree on the number of interactions they need to perform to reach a state, the other tries to establish whether components can agree on the order in which they cooperate. We combine our pairwise-analysis and these two global techniques in the *PairStatic* framework. This framework is reasonably scalable and it improves on Pair as it can capture some global invariants the system maintains to show deadlock freedom. For instance, PairStatic can capture invariants of some systems behaving like a systolic array or implementing a token mechanism. This work has been partially presented in [AGRR16b].

We go further on our analysis of token-based systems. Since this mechanism is so widely used in the context of distributed and concurrent systems, in Chapter 5, we propose two additional techniques aimed at capturing and using token invariants as reachability approximations. These techniques try to detect whether a system implements a token structure. They try to assign to each component state the number of tokens the component holds at that point. While one technique detects conservative token structures, where the interaction of components can cause tokens to be exchanged but not destroyed or created, the other detects existential structures, where tokens can be created and destroyed but not completely annihilated from the system. A token structure naturally leads to a token invariant that can be used as a reachability approximation. For instance, in the conservative case, we can count how many tokens are held by components in the initial state and, because tokens are conserved, this number must remain the same for every reachable state. Therefore, system states for which the sum of tokens does not add up to this initial value have to be unreachable. We combine our local-analysis and these two techniques in the *PairToken* framework. Unsurprisingly, this framework fares very well in proving deadlock freedom for systems that implement token mechanisms. It can even detect some useful token structures for systems that do not obviously implement a token mechanism. This work has been partially presented in [AGRR17a].

Unlike traditional approximate approaches, which strive for conditions that can be checked in polynomial time, our frameworks tackle *NP*-hard problems. Hence, there is no known polynomial-time algorithm that can detect deadlock states lying within our reachability approximations. Nevertheless, there have been some remarkable advances in proposing procedures to solve the propositional-satisfiability (SAT) and the satisfiability-modulo-theories (SMT) problems. So, we use modern SAT and SMT solvers to create efficient implementations for our frameworks. Also, our reachability

approximations can be reasonably straightforwardly encoded into SAT and SMT constraints.

Our reachability approximations give rise to useful frameworks for deadlock analysis and a natural question that arises is whether they can be used to check other properties. Our approximations are in no way tied to the analysis of deadlocks. So, they can be soundly used to check other properties. Still, we have to make sure that the verification frameworks they create are efficient and reasonably precise.

In Chapter 6, we investigate the use of our approximations in checking *static properties*. A static property is described in terms of the immediate behaviour of the system, i.e. the events (not) available at a system state. For instance, deadlock freedom, which is a static property, can be cast as "no system state can offer no events". Other examples of static properties are ensuring mutual exclusion and, generally, that some marked error state cannot be reached. Our approximations are combined into the *StaticProperty* framework to check static properties. This framework is able to show static properties for some non-trivial systems and it can do so quite efficiently. This work has been partially presented in [AGRR17b].

Static properties are unable to, simply, capture path-related behavioural properties; they cannot naturally capture, for instance, that a state/event must happen after some other state/event. To cope with that, in Chapter 7, we propose the *RefinementChecking* framework. It relies on an adapted watchdog-based refinement-checking approach powered by our approximations to analyse CSP's traces and failures refinement expressions. We also show how some relevant non-trivial expression can be effectively tackled by this framework.

1.1 Related Work

Local analysis has been used in many verification frameworks for different contexts and types of concurrency [RD87, BR91, AOS+14, ASW14, OAR+16, AC05, ABB+13, LMC11, Mar96]. The work in [RD87, BR91] introduces a theory of deadlocks based on the notion of an *ungranted request*, that is, a wait-for dependency between components. This work introduces a proof rule based around the *fundamental principle*: under reasonable assumptions about the system, a cycle of ungranted requests is a necessary condition for a deadlock. So, absence of such cycles demonstrates deadlock freedom. This proof rule provides a mathematical tool that can be manually used to show that a system is deadlock free.

The work in [OAR+16] introduces a semi-automatic framework to systematically design deadlock-free systems. It proposes a set of composition rules that only allow components to be composed if they interact properly. Moreover, it introduces a set of refinement expressions that can automatically check whether components interact properly. This framework, however, cannot efficiently tackle systems that have a cyclic communication topology. To cope with that, the work in [AOS+14, ASW14] introduces a set of design patterns that can be used to construct arbitrary-topology systems. The shortcoming of this approach is that there is only a handful of patterns that can be used to construct systems. It also provides a set of refinement expressions to automatically check that a system implements a pattern.

In [AC05, ABB+13, ABB+18, LMC11, Mar96], fully-automated approximate techniques for deadlock freedom are introduced. The work in [AC05] proposes a method for analysing syntactically-restricted shared-variable concurrent programs, whereas the framework in [ABB+13, ABB+18] adapts it to a more general setting meant to describe component-based message-passing systems. [LMC11] proposes a method for architecturally-restricted component-based systems interacting via message passing, and [Mar96] proposes a method for syntactically-restricted message-passing concurrent systems. All these frameworks are based around the fundamental principle so they use local analysis to prove the absence of cycles of ungranted requests. From analysing individual and pairs of components, they construct an ungranted-requests graph and show that such a cycle cannot arise in any conceivable state of the system.

To discuss in more detail how such techniques work, we present the *SDD* (State Dependency Digraph) framework, developed by Martin in [Mar96]. The original motivation for the work in this thesis was to create an improved version of this framework. Martin's analysis of SDDs is one of the most general prior approaches using local analysis to check deadlock freedom. It constructs a dependency digraph based on the analysis of pair of components, which is later checked for absence of cycles. The digraph has a node for each state of each component, and an edge from a state to another if and only if, considering only the pair of components, this pair of states is reachable and the first component is willing to interact with the second but they cannot coordinate on this interaction; this constitutes an ungranted request. This method is very efficient. The digraph can be constructed in quadratic time on the size of components, and it can be shown to have no cycles in linear time using a modified *depth-first-search*.

The efficiency of such frameworks comes with a price: the use of cycles of dependencies to approximate deadlocks can be imprecise in several ways. Firstly, a cycle might

not be consistent with basic sanity conditions such as it must have a single node per component (after all no component can be in two different states in a single deadlock). Secondly, a cycle is only partially consistent with the local reachability and local blocking information derived from the analysis of pairs of components. Note that only adjacent elements in the cycle are guaranteed to be pairwise reachable and pairwise blocked. So, there may be some local information of non-adjacent component states that could eliminate some cycle but it is not used. Finally, a cycle, as a necessary condition, is bound to arise in some deadlock-free systems. Thus, in such cases, this framework is ineffective. The reason why these sources of imprecision are not addressed is that these methods look for polynomially checkable conditions for guaranteeing deadlock freedom and tackling any of these sources of imprecision is likely to make the problem of finding a candidate in the dependency digraph *NP*-hard.

Lazy reachability is an exact approach that is based on local analysis [JL16]. It begins analysing the behaviour of a small set of components, and this set is incrementally augmented until either the property is shown or a counter example is found. This approach works well when the property being tested can be demonstrated using local analysis. Otherwise, it ends up analysing the entire system and suffering with the state-space explosion problem.

Pure local-analysis techniques cannot prove a property that depends on some global invariant of the system. To circumvent this issue, many frameworks combine approximate global analysis with local analysis.

Martin proposed two extensions to the SDD that implement approximate global analysis: the *CSDD* (Coloured State Dependency Digraph) and *FSDD* (Flashing State Dependency Digraph) [Mar96] frameworks. They extend 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 reaching back its initial state as a way to determine that the component has completed a cycle of interactions. For each edge in the dependency digraph, it analyses how pairs of components behave to establish whether, when reaching this pair of component states, components have performed the same number of cycles, or one has performed more cycles than the other, or neither of these two possibilities has arisen. This information eliminates some dependency cycles: any cycle where components have performed more cycles than their predecessors represents an impossibility.

Similar to CSDD, for each edge, FSDD establishes whether, when this pair of states is reached, the first component has interacted at least once and with the second component more recently than with any other component or not. This information is used to eliminate dependency cycles where components have more recently communicated with their predecessor. Again, this is an impossibility.

These two methods were proposed to prove deadlock for classes of systems with cyclically-interacting components, which usually implement some global mechanism to avoid deadlocks. 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.

Data flow analysis is an approach used to approximate the behaviour of sequential and concurrent systems [RS90, CK94, DCCN04, BK11, BCC+03]. To some extent, CSDD and FSDD use, in an ad-hoc way, data-flow analysis to calculate the extra reachability information added to dependencies.

The work in [CK94] 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 [DCCN04] 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. Then, 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 the counter-example generated by the

framework, the user identifies whether it is spurious and, if so, designs some constraint to eliminate it.

Verification frameworks often use invariants as compact abstractions for the behaviour of concurrent and distributed systems [Lam77, AFDR80, BL99]. Data-flow analysis is one method to calculate global invariants but others exist.

D-Finder 2 is a deadlock-checking tool that handles component-based systems described in the BIP notation [BGL+11, BBL+16]. 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 [Mur89]. 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.

Most approximate frameworks verify a specific property and they are based on conditions that are inherent to these properties. We described some frameworks checking deadlock freedom and there exist frameworks checking for livelock freedom [OPRW13, FOSC16] and determinism [OCS17]. These frameworks are quite difficult to adapt for other properties and, when they are adapted, there might be a considerable loss in terms of both verification speed and precision. We discussed frameworks that check for more general properties such as [CK94] that tests for properties cast as reachability goals, whereas [DCCN04] test for properties captured by a quantified regular expression.

1.2 Outline

This thesis is organised as follows. Chapter 2 introduces the background material upon which our work is based. It introduces supercombinator machines and CSP, formal notations we use to reason about concurrent and distributed systems, and the basics of SAT and SMT solving, the procedures used to implement our frameworks. Chapter 3 formally defines the notion of local analysis. This chapter also introduces a reachability over-approximation based on this notion and a framework, called Pair, that checks deadlock-freedom using this approximation.

Chapter 4 and 5 introduce global-analysis techniques to over-approximate reachability. Chapter 4 presents some synchronisation-analysis techniques. They use abstract-interpretation concepts, which are usually part of a data-flow-analysis framework, to compute component invariants that are then combined to approximate reachability. We also propose some abstraction techniques to improve on component analysis. This chapter also presents how these techniques are employed by the PairStatic framework to verify deadlock freedom. Chapter 5 proposes some techniques based on token structures and invariants. It presents a method that allow for detecting and deriving such invariants. This chapter also introduces PairToken, a framework that uses these approximations to check deadlock freedom.

Chapter 6 investigates the use of these approximations in the analysis of static properties, namely, a class of properties that can be formulated as "no bad combination of events can be offered (refused)". Chapter 7 extends this investigation for the analysis of CSP's traces and stable-failures refinement specifications. Finally, Chapter 8 presents our final remarks about this work.

Chapter 2

Background

In this chapter, we introduce some notations and tools that are used in this work. We introduce CSP, a formal notation to model concurrent and distributed systems; FDR4, a model checker to the CSP notation; and the basics about SAT and SMT solvers. This work only tangentially depends on the CSP notation, for our frameworks are founded on supercombinator machines: the combinator-based structures used by FDR4 to capture CSP systems. As the frameworks we propose in this work are implemented using SAT and SMT solvers, we introduce how these solvers work and a few of the main features that make them tackle hard problems.

2.1 CSP

Communicating Sequential Processes (CSP) [Hoa85, Ros10] is a formal notation used to model concurrent and distributed systems where processes/components interact by exchanging messages. This process algebra has been successfully used to formally analyse many systems and it is also the formalism upon which several techniques are based [Low96, CR99, MS01, GMR+03, RW06, CGRX06, RSM09, SNM09, OPRW13, AOS+14]. A significant part of this success is due to strong tools that automatically verify CSP models. FDR4 [GRABR14], the standard model checker for CSP, PAT [SLDP09] and ProB [LB05] are examples of such tools.

This work is based on the combinator-style operational semantics of CSP, introduced in Section 2.2, and as such it depends very little on CSP itself. So, we only present a subset of the CSP language that we judge helpful in understanding this work. Our intent is to provide enough background so one can understand how CSP terms (or, possibly, terms of other formalisms) are (could be) treated in this combinator-based operational semantics. For a full account of CSP, the reader is referred to [Ros10].

2.1.1 Syntax

In CSP, concurrent and distributed systems are built by a combination of sequential processes using high-level parallel operators. Sequential processes are generally built using the following five constructs and CSP's basic processes $STOP$ and $SKIP$. $STOP$ is the deadlocked process, whereas $SKIP$ represents successful termination. In the following, P, Q represent arbitrary CSP processes and a, b, c, d arbitrary CSP events.

Prefixing

Prefixing ($\rightarrow$) is the most fundamental operator in CSP. The process $a \rightarrow P$ first communicates/offers event a and then behaves as process P. This event can be replaced by an input communication $ch?e$ which means that the process is initially ready to receive any data associate with channel ch.

External choice

The external-choice process $P \Box Q$ offers initially the events offered by both P and Q. By performing one of these events, either process P or Q is chosen, depending on which of them offered the performed event. For instance, consider $P = a \rightarrow c \rightarrow STOP$ and $Q = b \rightarrow d \rightarrow STOP$, process $P \Box Q$ first offers events a and b. If a is performed, P is chosen and event c is offered. In the same way, if b is performed, it chooses Q and offers event d.

Internal choice

Unlike the external choice where a process can be selected by choosing an event, the internal-choice process $P \sqcap Q$ internally (i.e. non-deterministically) decides to behave as either P or Q.

Sequential composition

The sequential-composition process $P\,;Q$ behaves as P until it successfully terminates, then it behaves as Q. For instance, process $P\,;\,Q$, where $P = a \rightarrow SKIP$ and $Q = b \rightarrow STOP$, initially offers event a, then event b after which it stops/deadlocks.

Recursion

Recursion can be achieved by using a reference to the process in its own definition. For instance, $P = a \rightarrow P$ performs a and then recurses to P.

Distributed alphabetised parallelism

After modelling sequential processes using these basic processes and operators, one can create a parallel system using the replicated-alphabetised-parallel operator.

The replicated-alphabetised-parallel process $P = \|_{i=0}^{N} (A_i, C_i)$ puts processes C_i in parallel considering their respective alphabets A_i. So, process C_i can only offer events in A_i. If event a is shared by $C_j, \ldots, C_k$ (i.e. $a \in A_j \cap \ldots \cap A_k$), P can only offer it if all components $C_j, \ldots, C_k$ are simultaneously offering it; all these processes synchronise on the event a.

Hiding

Finally, the hiding operator ($\backslash$) is commonly used as a means to abstract away events. The process $P \setminus S$ offers the *invisible* event τ instead of events in S for P.

We use Milner's scheduler as a running example to demonstrate the concepts we introduce in this section. This system is a token ring where a token rotates amongst components and the component that possesses the token is scheduled to work.

Example 2.1. For a given $N \in \mathbb{N}$, Milner's scheduler with N components is modelled by the following process $SC_N = \|_{i=0}^{N-1} (A_i, C_i)$ where:

- $C_0 = a_0 \rightarrow c_1 \rightarrow b_0 \rightarrow c_0 \rightarrow C_0$;
- $C_i = c_i \rightarrow a_i \rightarrow c_{i\oplus 1} \rightarrow b_i \rightarrow C_i$;
- $A_i = \{a_i, b_i, c_i, c_{i\oplus 1}\}$;
- $\oplus$ is addition modulo N.

In this example, events c_i represent that component $C_{i\ominus 1}$ ($i \ominus 1$ is subtraction modulo N) has passed the token to C_i, events a_i represent component C_i's work, and events b_i signal that component C_i is in a idle state, waiting for the token. So, in this system, component C_0 has the token initially, and whenever a component C_i possesses the token it first works, then it passes the token along to $C_{i\oplus 1}$ and goes into an idle state, where it waits for the token to return to it. $\blacksquare$

2.1.2 SOS-style operational semantics

The operational semantics of CSP associates each syntactic process with a labelled transition system (LTS) representing its behaviour. As a convention, let $\mathcal{E}$ denote the finite universal set of *visible* events, $\tau \notin \mathcal{E}$ the *invisible* event, and $\checkmark \in \mathcal{E}$ the termination signal.

Definition 2.1. A labelled transition system 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;
- $\hat{s} \in S$ is the starting state.

Operational semantics have been traditionally represented using Plotkin's SOS (Structured Operational Semantics) framework [Plo81]. This framework proposes inference rules, called *firing rule*, that can be used to deduce the events a CSP process initially offers and the process resulting from performing each of these events.

Figure 2.1 shows some of CSP's firing rules. These inference rules have three elements: a *proviso* above the horizontal line, a *side condition* to the right-hand side of the line, and a *consequence* below the line. If the proviso and side condition hold, the consequence can be deduced. An empty proviso or side condition means they are always satisfied. For instance, Rule 2.1 captures how prefixing processes behave: $a \rightarrow P$ can perform event a and then it behaves as P. Rules 2.6, on the other hand, capture how the replicated-alphabetised-parallel process behaves. The first rule captures that a transition involving a visible event happens when all component processes sharing this event agree on performing it, while the second describes how components can individually progress by performing the invisible event τ. Note how these rules describe the behaviour of a CSP construct based on its operands, hence the term *structured*. We illustrate the use of this rules using Example 2.1; in Figure 2.2, we present the LTSs for component processes C_0 and C_1, and the system SC_2.

2.1.3 Denotational Semantics

CSP has also been given a number of denotational (behavioural) models. In these models, a process is associated with a set of behaviours it exhibits. In this work, we are interested in two denotational models: the traces and stable-failures models.

These denotational models are equipped with a set of clauses, one for each CSP construct, so that one can calculate the behaviours of a process. In this work, however, as our main focus is on the operational semantics, we omit these clauses and, instead, demonstrate how behaviours can be extracted based on the LTS of a process.

$$\frac{}{a \rightarrow P \xrightarrow{a} P} \tag{2.1}$$

$$\frac{P \xrightarrow{a} P'}{P \mathbin{\Box} Q \xrightarrow{a} P'}\ a \neq \tau \qquad \frac{Q \xrightarrow{a} Q'}{P \mathbin{\Box} Q \xrightarrow{a} Q'}\ a \neq \tau$$
$$\frac{P \xrightarrow{\tau} P'}{P \mathbin{\Box} Q \xrightarrow{\tau} P' \mathbin{\Box} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \mathbin{\Box} Q \xrightarrow{\tau} P \mathbin{\Box} Q'} \tag{2.2}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q} \tag{2.3}$$

$$\frac{P \xrightarrow{a} P'}{P\,;Q \xrightarrow{a} P'\,;Q}\ a \neq \checkmark \qquad \frac{P \xrightarrow{\checkmark} P'}{P\,;Q \xrightarrow{\tau} Q} \tag{2.4}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus S \xrightarrow{a} P' \setminus S}\ a \notin S \qquad \frac{P \xrightarrow{a} P'}{P \setminus S \xrightarrow{\tau} P' \setminus S}\ \ a \in S \tag{2.5}$$

For $J \subseteq \{0 \dots N\}$ and $\overline{S} = \{0 \dots N\} - S$:

$$\frac{\forall\, j \in J \bullet C_j \xrightarrow{a} C'_j \wedge a \in A_j \quad \forall\, k \in \overline{J} \bullet C_k = C'_k \wedge a \notin A_k}{\big\Vert_{i=0}^{N} (A_i, C_i) \xrightarrow{a} \big\Vert_{i=0}^{N} (A_i, C'_i)}\ a \neq \tau$$
$$\frac{C_i \xrightarrow{\tau} C'_i \qquad \forall\, k : \overline{\{i\}} \bullet C_k = C'_k}{\big\Vert_{i=0}^{N} (A_i, C_i) \xrightarrow{\tau} \big\Vert_{i=0}^{N} (A_i, C'_i)}\ i \in \{0 \dots N\} \tag{2.6}$$

Figure 2.1: SOS rules

2.1.3.1 Traces

In this model, a process is identified with the set of its *traces*. A trace is a sequence of visible events a process can engage on. We can capture the traces of a process based on its LTS. Before we do so, we introduce the following notation:

Definition 2.2. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS, $s, s' \in S$ states, $a \in \Sigma$ an event, and $\langle a_1, \dots, a_n \rangle, tr \in \Sigma^*$ sequences of events.

- $s \xrightarrow{a} s'$ if and only if $(s, a, s') \in \Delta$.

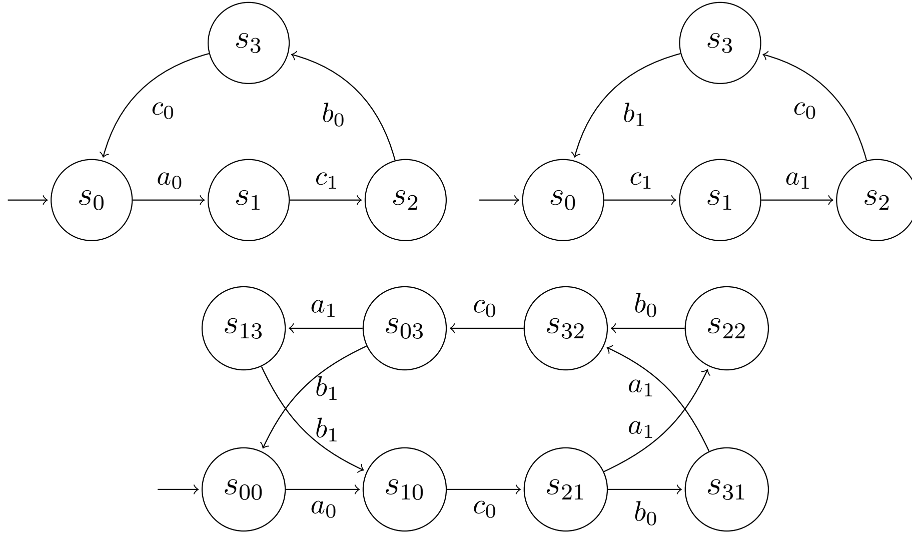

Figure 2.2: LTSs of components C_0 and C_1 above that of SC_2.

- $s \xrightarrow{\langle a_1,\ldots,a_n\rangle} s'$, namely, there is a path from s to s' with a sequence of events $\langle a_1, \ldots, a_n\rangle$ if and only if 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 $s_n = s'$.

- $s \overset{tr}{\Longrightarrow} s'$, namely, tr is a trace leading L from s to s' if and only if there exist a path $s \xrightarrow{\langle a_1,\ldots,a_n\rangle} s'$ such that tr is the sequence of events resulting from removing all τ-occurrences from $\langle a_1, \ldots, a_n\rangle$.

So, the traces of a LTS are the ones it can engage on from its initial state.

Definition 2.3. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS. Its traces are given by:

$$traces(L) \mathrel{\widehat{=}} \{tr \mid \exists\, s \in S \bullet \hat{s} \overset{tr}{\Longrightarrow} s\}$$

For instance, the traces of components C_0 and C_1, in Example 2.1, can be calculated from theirs LTSs in Figure 2.2:

$$\begin{aligned} traces(C_0) &= \{\langle\rangle, \langle a_0\rangle, \langle a_0, c_1\rangle, \langle a_0, c_1, b_0\rangle, \langle a_0, c_1, b_0, c_0\rangle, \langle a_0, c_1, b_0, c_0, a_0\rangle, \ldots\} \\ traces(C_1) &= \{\langle\rangle, \langle c_1\rangle, \langle c_1, a_1\rangle, \langle c_1, a_1, c_0\rangle, \langle c_1, a_1, c_0, b_1\rangle, \langle c_1, a_1, c_0, b_1, c_1\rangle, \ldots\} \end{aligned}$$

2.1.3.2 Stable failures

In this model, a process is identified with its sets of *stable failures* and traces. Stable failures capture information about the events a process refuses to perform (is unable to perform). We capture such sets of refused events using the following notation.

Definition 2.4. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS, $s \in S$ a state, and $X \subseteq \mathcal{E}$ a set of events. $s\ \mathit{ref}\ X$ denotes that L refuses events in X at state s, whereas $\mathit{initials}(s)$ gives the events available at s.

- $s\ \mathit{ref}\ X$ holds if and only if $X \cap \mathit{initials}(s) = \emptyset$
- $\mathit{initials}(s) = \{a \mid s \xrightarrow{a} \wedge\ a \in \mathcal{E}\}$
- $s \xrightarrow{a}$ holds if and only if there exits a state $s' \in S$ such that $s \xrightarrow{a} s'$

A stable failure is a pair containing a trace and a set of events the system can refuse after this trace. It is *stable* because it does not record refusals information for states in which the process is *unstable*, that is, it can offer a τ event.

Definition 2.5. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS. Its stable failures are:

$$\mathit{failures}(L) \mathrel{\widehat{=}} \{(tr, X) \mid tr \in \Sigma^* \wedge X \subseteq \mathcal{E} \wedge \exists\, s \in S \bullet \hat{s} \overset{tr}{\Longrightarrow} s \wedge s \overset{\tau}{\not\longrightarrow} \wedge\ s\ \mathit{ref}\ X\}$$

For instance, the stable failures of components C_0 and C_1, in Example 2.1, can be calculated from theirs LTSs in Figure 2.2. We use $(tr, X_{\mathbb{P}})$ to concisely represent all pairs of stables failures (tr, X') where $X' \subseteq X$.

$$\mathit{failures}(C_0) = \{(\langle\rangle, \{c_0, c_1, b_0\}_{\mathbb{P}}), (\langle a_0\rangle, \{a_0, c_0, b_0\}_{\mathbb{P}}), (\langle a_0, c_1\rangle, \{a_0, c_0, c_1\}_{\mathbb{P}}), \ldots\}$$
$$\mathit{failures}(C_1) = \{(\langle\rangle, \{a_1, c_0, b_1\}_{\mathbb{P}}), (\langle c_1\rangle, \{c_0, c_1, b_1\}_{\mathbb{P}}), (\langle c_1, a_1\rangle, \{a_1, c_1, b_1\}_{\mathbb{P}}), \ldots\}$$

2.1.3.3 Refinement relations

The reason for providing precise semantics to processes is to enable the precise analysis of their behaviour. Multiple behavioural models, capturing different behavioural aspects, are provided so the specifier (i.e., user) of the CSP notation can choose the model that best fits their needs. For instance, the traces model provides an unambiguous way to describe the visible sequence of events a system can engage on. So, this model can be used to prove that a system cannot perform some sequence of events that would constitute some sort of *bad* behaviour. This model, however, does not record information about the refusal of events and, as such, it cannot show that a system might deadlock or that it is non-deterministic; two processes that perform the same set of traces but with one deadlocking or performing some non-deterministic behaviour cannot be picked apart by this model. Hence, the need for a finer model such as the stable-failures one. As this model records both traces and stable failures of a process, one can check for deadlocks and non-determinism using the additional information recorded in the stable failures of a process.

Traditionally in CSP, systems are analysed, and properties captured, using *refinement expressions*. A refinement expression involves a specification process P, an implementation process Q and a *refinement relation* $\sqsubseteq$. So, an expression $P \sqsubseteq Q$ holds if Q satisfies the behavioural specification P, where this behavioural satisfaction notion is defined by $\sqsubseteq$. Both traces and stable-failures models offer a refinement relation. These relations entail different behavioural obligations, which are, obviously, restricted to the information available for each model. In the following, we introduce these two refinement relations. As we take an operational-semantics perspective on CSP, we define these relations in terms of LTSs instead of the traditional process-based definition.

The traces-model refinement relation, given by $\sqsubseteq_T$, holds if the implementation's traces are a subset of the specification's.

Definition 2.6. Let L_{sp} be a specification system and L_I an implementation.

$$L_{sp} \sqsubseteq_T L_I \text{ if and only if } \mathit{traces}(L_I) \subseteq \mathit{traces}(L_{sp})$$

This relation (and behavioural model) is useful for capturing safety properties of the form: "a system does not engage in a bad sequence of events". For an implementation L_I, this sort of property can be captured by designing a specification L_{sp} that can perform all *good* traces and then ensuring that $L_{sp} \sqsubseteq_T L_I$ holds.

For instance, we can use the following refinement expression to check whether the system SC_2 in Example 2.1 only allows the token to alternate between C_0 and C_1, that is, it checks that the events c_0 and c_1 arise in the following order $c_1, c_0, c_1, c_0, \ldots$.

$$L_{sp} \sqsubseteq_T L_{SC_2 \setminus \{a_0, a_1, b_0, b_1\}} \tag{2.7}$$

In this refinement expression, $L_{SC_2 \setminus \{a_0, a_1, b_0, b_1\}}$ is the LTS of process $SC_2 \setminus \{a_0, a_1, b_0, b_1\}$ and L_{sp} is the LTS of process $P = c_1 \rightarrow c_0 \rightarrow P$. We point out that P can only perform the sequence of good events involving c_0 and c_1, and $SC_2 \setminus \{a_0, a_1, b_0, b_1\}$ is the SC_2 with the events a_0, a_1, b_0, b_1, which are not relevant for this property, abstracted away.

The stable-failures refinement relation, given by $\sqsubseteq_F$, holds if the implementation's traces and stable failures are subsets of the specification's traces and stable failures.

Definition 2.7. Let L_{sp} be a specification and L_I an implementation.

$$L_{sp} \sqsubseteq_F L_I \text{ if and only if } \mathit{traces}(L_I) \subseteq \mathit{traces}(L_{sp}) \text{ and } \mathit{failures}(L_I) \subseteq \mathit{failures}(L_{sp})$$

This relation (and behavioural model) can capture finer properties if compared to its traces counterpart. Roughly speaking, this relation captures properties of the form: "not only a system does not engage in a bad sequence of events, but it must refuse less (or, accept more) events after an allowed trace". This relation, in particular, requires the implementation to be more deterministic than the specification. As for traces, this sort of property can be captured by combining the stable-failures refinement relation with a specification that can perform all *good* traces and that accepts the minimum desired set of events after each valid trace.

For instance, let us replace the refinement relation in the refinement expression in 2.7 to get the following refinement expression.

$$L_{sp} \sqsubseteq_F L_{SC_2 \setminus \{a_0, a_1, b_0, b_1\}}$$

This expression holds if the system SC_2 in Example 2.1 only allows the token to alternate between L_0 and L_1 and does not refuse c_0 after c_1 and c_1 after C_0. To illustrate the different between these two expressions (and models), we point out that the deadlocked system $Q = STOP$ is a satisfying implementation to the traces version of this refinement expression, whereas the same implementation would not satisfy this stable-failures version.

One might wonder what is the purpose of the traces model given that the stable-failures is a finer model that allows for strictly better analysis of processes. The main reason is that (automatically) checking a traces refinement expression is easier than checking a stable-failures one, as more behavioural aspects are involved in the latter.

2.2 FDR4

In this section, we introduce how FDR4 [GRABR14] implements CSP's operational semantics and a few of its main features. Instead of using the traditional SOS framework, FDR4 uses a *combinator-based* approach to implicitly capture the LTS of a system. We also discuss how refinement checking is carried out by FDR4 and two techniques, namely, *partial order reduction* and *compression*, aimed at speeding up the analysis of systems by trying to reduce the state space to be analysed.

2.2.1 Combinator-style operational semantics

Instead of using the SOS rules to explicitly generate the LTS of a system, FDR4 relies on a *combinator-based* operational semantics that represents systems as *supercombinator machines*. This representation implicitly captures the LTS of a system in a concise way.

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

- $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ is a sequence of component LTSs such that $n \geq 1$;
- $\mathcal{R}$ 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.
  - $a \in \mathcal{E} \cup \{\tau\}$ is the event the supercombinator machine performs.

We use the square-bracketed set $[(i_1, a_1), \ldots, (i_m, a_m)]$ as an alternative way to represent the tuple $(e_1, \ldots, e_n)$ where $e_i = a$ if the pair (i, a) belongs to the set and $-$ otherwise. Note m does not need to be equal to n. We use $[i, a]$ as a shorthand for the single-pair set $[(i, a)]$, and set operations between these sets create a tuple based on the resulting set.

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

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

Definition 2.9. A *supercombinator machine* $(\mathcal{L}, \mathcal{R})$ with n components is triple disjoint if and only if for all pairs $(e, a) \in \mathcal{R}$, e is triple disjoint, that is, at most two components participate in a rule.

$$\begin{array}{l} triple_disjoint(e) = \\ \qquad \forall\, i, j, k : \{1 \ldots n\} \mid i \neq j \wedge j \neq k \wedge i \neq k \bullet e_i = - \vee e_j = - \vee e_k = - \end{array}$$

Triple disjointness is also a soft restriction; a single-format machine can also be translated in polynomial time into a triple-disjoint single-format machine. This translation is more complicated than the previous one and involves the creation of new components, one per rule of the original machine, to emulate rule application. Therefore, a general supercombinator machine can be efficiently converted (i.e. in polynomial time) into a single-format triple-disjoint one, which can then be input to the verification frameworks we propose. We reinforce, however, that this translating should hinder the precision of our frameworks. So, our frameworks are more suited to tackle machines that are naturally described in this form. We also point out that many supercombinator machines are naturally triple disjoint. Many distributed systems and component-based systems are naturally modelled using pairwise communication. Moreover, triple disjointness is also a restriction imposed by many notations and frameworks that are similar to ours [Ram11, Mar96, AC05, FOSC16, AOS+14, ASW14]. Our translation could, potentially, be leveraged by these frameworks to lift their requirement for triple-disjointness.

To illustrate the notion of a supercombinator machine, we provide a machine definition that captures system SC_2 in Example 2.1.

Example 2.2. Let L_1 and L_2 be the LTSs of components C_0 and C_1, respectively, as depicted in Figure 2.2. The system SC_2 presented in Example 2.1, which models Milner's scheduler with 2 components, can be represented by the supercombinator machine $\mathcal{S}_{SC_2} = \{\langle L_1, L_2 \rangle, \mathcal{R}\}$, where the rules in $\mathcal{R}$ require components to synchronise on shared events:

$$\mathcal{R} = \{((a_0, -), a_0), ((b_0, -), b_0), (-, a_1), a_1), ((-, b_1), b_1), ((c_0, c_0), c_0), ((c_1, c_1), c_1)\}$$

■

A supercombinator machine can be seen as an implicit representation of a system in the sense that it *induces* a LTS representing its behaviour.

Definition 2.10. 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:

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

In this work, we use *system state* (*component state*) to designate a state in the system's (component's) LTS. From now on, we refer to a system and its supercombinator machine interchangeably. 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 2.11. For induced LTS $(S, \Sigma, \Delta, \hat{s})$, state $s \in S$ is reachable if and only if $reachable(s)$ holds, where $reachable(s) = \exists p : \Sigma^* \bullet \hat{s} \xrightarrow{p} s$.

FDR4 implements a *compilation* procedure that translates a parallel CSP process into a supercombinator machine. Informally, for a replicated-alphabetised-parallel CSP process, it uses SOS rules to explicitly create the LTSs of component processes and it relies on the analysis of alphabets and shared events to create rules. If some processes share an event, the compilation procedure creates a rule that requires the participation of each of these process by performing this event and that yields this event as the system's top-level event. As invisible events are not shared, each component that can perform an invisible event would have a rule in which it individually participates on and yields τ as top-level event. These combinator-style rules, as much as the SOS Rules 2.6 in Figure 2.1, capture the concurrent message-passing behaviour of parallel CSP processes.

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

2.2.2 Property and refinement checking

FDR4 is a fully-automatic verification tool that checks properties about CSP systems. In this work, we are concerned with three problems (properties) that can be tackled (verified) by FDR4: *deadlock-freedom*, *traces-refinement* and *failures-refinement* problems. In this section, we introduce these problems, discuss how FDR4 tackles them, and show they are all *PSPACE*-complete. Intuitively, the exploration of induced LTSs can be seen as a cause for the *PSPACE* complexity of these problems. This exploration tends to be severely hindered by the *state-space-explosion problem*, i.e. with the linear increase in the number of components in the machine its induced LTS grows exponentially. Currently, there is no known algorithm that can solve such problems in polynomial time, and the common belief is that there is none; current algorithms employed to solve *PSPACE*-complete problems take exponential time.

A problem $\mathcal{P}$ is C-complete, where C is a complexity class, if and only if $\mathcal{P}$ is both a member of C and it is also C-hard. $\mathcal{P}$ is C-hard if all problems in C can be reduced in polynomial time to $\mathcal{P}$. *PSPACE*, *NP*, and *P* are examples of complexity classes; *PSPACE* encompass the problems that can be decided using polynomial space by a Turing machine, *NP* the problems that can be decided in by a non-deterministic Turing machine in polynomial time, and problems in *P* can be decided in polynomial time by a deterministic Turing machine. It is widely known that $\mathit{PSPACE} \supseteq \mathit{NP} \supseteq P$; whether the reverse containments hold, however, is still an open question – these containments actually amount to showing whether $P = \mathit{NP}$ and $\mathit{NP} = \mathit{PSPACE}$, two very hard questions that have been open for decades [Sip12]. Showing that a problem is C-hard is usually interpreted as a sort of lower-bound result, that is, this problem belongs to a class C' such that $C \subseteq C'$. On the other hand, showing membership to C is usually interpreted as providing an upper bound, namely, if a problem belongs to C, it might even belong to some complexity class C' such that $C' \subseteq C$. In our complexity analyses, we use high-level (pseudo-code-like) description of Turing machines [Sip12].

To show *PSPACE*-hardness for these problems, we rely on reductions from the *single-component traces-refinement problem*. For a specification LTS L_{sp} and an implementation LTS L, both of which are finite, this problem asks whether $L_{sp} \sqsubseteq_T L$ holds. This problem has been shown to be *PSPACE*-complete in [KS90]:

Lemma 2.12. *Given a specification LTS L_{sp} and an implementation LTS L_I, the problem of checking that $L_{sp} \sqsubseteq_T L_I$ holds is* PSPACE*-complete.*

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

Lemma 2.13. NPSPACE = PSPACE.

The widely-known result presented in Lemma 2.12 assumes that the input system is described by a LTS depicting its behaviour. By assuming this input representation, this lemma is somewhat ignoring the state-space explosion problem. In our context, the (induced) LTS of a concurrent system can be exponentially large on the size of its supercombinator machine description. In this thesis, we *are* particularly interested in understanding how the process of constructing/analysing an *induced* LTS (and the issuing state-space explosion problem) affects verification. So, the problems we investigate assume that the system being analysed is described by a supercombinator machine instead of its induced LTS. By assuming this new input representation, we investigate a *new* set of problems and prove *new* results about them.

For the sake of decidability, we only consider supercombinator machines with a finite number of components, which are themselves represented by finite LTSs. We use the following definitions for the size of a supercombinator machine and a LTS.

Definition 2.14. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a LTS and $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ a supercombinator machine:

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

The first problem that we discuss is the deadlock-freedom problem. A system deadlocks when it reaches a state in which it becomes *blocked*, namely, unable to perform any further event. So, a system is deadlock free if no such state exists.

Definition 2.15. 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 $\neg \exists s : S \bullet deadlock(s)$ holds.

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

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

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

Theorem 2.16. *The deadlock-freedom-checking problem is* PSPACE-*complete.*

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

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

Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be the supercombinator machine we are trying to show deadlock free, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $L = (S, \Sigma, \Delta, \hat{s})$ its induced LTS . We call NTM Check in Algorithm 2.1 with input $(\mathcal{S}, \hat{s}, |S|)$, where $|S| = \prod_{i \in \{1 \ldots n\}} |S_i|$. If it accepts this input, $\mathcal{S}$ has a deadlock, if it rejects, $\mathcal{S}$ is deadlock free. A non-deterministic Turing machine accepts some input if some branch yields *accept*, and rejects it if all branches yield *reject*. Each *guess* (and recursive call to Check) creates a branch that has to store the supercombinator $\mathcal{S}$, a state s, a number n, and some extra space to calculate *blocked*(s) and *sucessor*(s), namely, the memory used is proportional to the size of $\mathcal{S}$. So, it uses polynomial space in the size of supercombinator machine $\mathcal{S}$. Note that the number of recursive calls (i.e. the depth of our computation tree) is bounded by $|S|$.

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

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

Let $L_{sp} = (S_{sp}, \Sigma_{sp}, \Delta_{sp}, \hat{s}_{ap})$, where $S_{sp} = \{s_1, \ldots, s_n\}$, be a specification LTS and L_I an implementation LTS with alphabet Σ_I. We assume, without loss of generality, that L_{sp} and L_I are τ-free and $\tau \notin \Sigma_I \cup \Sigma_{sp}$; we could run a τ-elimination procedure that runs in polynomial time on L_{sp} and L_I and creates a traces-equivalent τ-free LTS. We propose the creation of the supercombination machine $\mathcal{T}(L_{sp}, L_I)$, described

Algorithm 2.1 NTM to check for a deadlock for machine $\mathcal{S}$.

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

next, as a means of reducing the verification of single-component traces refinement between these LTSs to checking deadlock freedom for this machine.

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

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

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

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

Component CC, roughly speaking, reads an event the currently-being-visited implementation state can perform and tries to see if the determinised specification

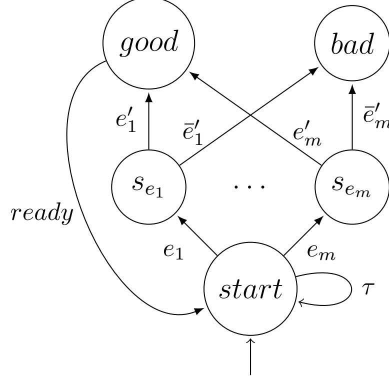

Figure 2.3: LTS sketch of component CC for $\Sigma_I = \{e_1, \ldots, e_m\}$.

counterpart state can perform it too. If so, it moves on to next pair of states to be visited. On the other hand, if the specification cannot perform this event it goes into a deadlock state, which leads the entire system into a deadlock. In this machine, in addition to the original events e, we use fresh events e' and $\bar{e}'$ to denote that the determinised specification can and cannot perform event e, respectively. CC definition is given next, and we provide a sketch of its graphical representation in Figure 2.3.

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

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

We use components $S_1, \ldots, S_n$ to represent a state of the determinised specification. Component S_i accounts for the behaviour of state s_i in L_{sp}. S_i's states *on* and *off* tell whether s_i is part of the determinised state currently being visited, whereas states s_e serve to update what the next determinised state should be based on the successors of s_i for event-e transitions in L_{sp}. We provide a definition for this component next, and a sketch of its graphical representation in Figure 2.4.

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

- $S = \{on, off\} \cup \{s_e \mid s_i \xrightarrow{e}\}$
- $\Sigma = \{ready, on_i\} \cup \{e, e', \bar{e}' \mid s_i \xrightarrow{e}\}$

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

Component C_i is a simple controller to S_i. It keeps information about whether this state should be part of the next determinised state being visited, namely, whether S_i should be "turned on". The definition for this component is given next. Also, we provide its graphical representation in Figure 2.5.

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

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

Finally, we describe, as follows, the rules that regulate the interaction between components in this machine. For convenience, we use our square-bracketed notation to conveniently represent an event tuple as a set of pairs. Moreover, we use the name of components to conveniently represent their position in a rule's event tuple: $S_i = 2i - 1$, $C_i = 2i$, $CC = 2n + 1$, $L_I = 2n + 2$.

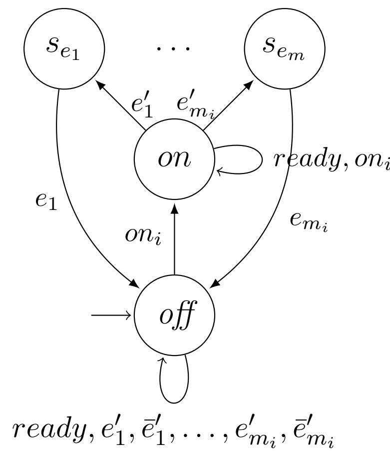

Figure 2.4: LTS sketch of component S_i with $\{e \mid s_i \xrightarrow{e}\} = \{e_1 \ldots e_{m_i}\}$

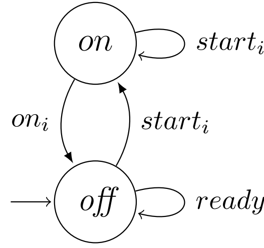

Figure 2.5: LTS of component C_i.

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

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

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

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

The other two problems that we tackle are the traces-refinement and the failures-refinement checking. Unlike traditional frameworks, when considering this problem, we assume that the specification LTS has already been *normalised* (i.e. determinised).

Traditional frameworks employ a normalisation procedure to determinise the specification, i.e. this procedure creates a *normal* LTS that is behaviourally equivalent to the specification. Normalising the specification facilitates the process of refinement checking. Instead of comparing an implementation state with possibly many specification states that can be reached via the same trace, the normalisation process guarantees that only one state of the normalised specification can be reached by a given trace. Normalising the implementation, however, does not give any evident advantage.

Definition 2.17. A *normal* LTS has the property that every trace leads to a single state. That is, for any trace tr of the normal LTS, if $\hat{s} \xrightarrow{tr} s$ and $\hat{s} \xrightarrow{tr} s'$ then $s = s'$, where $\hat{s}$ is the initial state of the normal LTS, and s and s' two of its states. Also, the LTS must not have τ transitions.

With the assumption that the specification has already been normalised, we purposely disregard the (possibly substantial) complexity of normalising the specification. The reason for this assumption is two-fold. Firstly, we use it to show that refinement checking is highly complex even if we disregard normalising the specification. Secondly, most practical specifications do not require a complex normalisation. So, this assumption should more accurately capture the complexity of checking conventional refinement expressions. In this work, we are interested in tackling the complexity arising from the potential blow up in the number of states of an induced LTS. So, our techniques make no effort in trying to tame the complexity of normalisation.

FDR4's main feature is, arguably, its ability to automatically check whether a refinement expression holds. For the traces behavioural model, we have the following refinement problem.

Definition 2.18. Given a supercombinator machine $\mathcal{S}_I$ and a normalised specification LTS L_{sp}, the traces-refinement problem asks whether $\mathcal{S}_I$'s induced LTS L_I refines L_{sp}, namely, whether $L_{sp} \sqsubseteq_T L_I$ holds.

FDR4 has a refinement-checking engine that is responsible to carry out such checks. After normalising the specification, it carries out the exploration of the product space of specification and implementation. This exploration goes through pairs of states (s_{sp}, s_I) such that s_{sp} is a state of L_{sp} and s_I is a state of L_I and they can be reached via the same trace. For each of these pairs, it tests whether s_I can perform a visible event that s_{sp} cannot, namely, whether $initials(s_{sp}) \not\supseteq initials(s_I)$ holds. If such a pair exists, the refinement expression does not hold. Otherwise, it does.

The problem of checking whether a traces-refinement expression holds is also *PSPACE*-complete. So, automatic verification techniques also tend to struggle to check traces-refinement expressions even for some systems with few components.

Theorem 2.19. *The traces-refinement problem is* PSPACE*-complete.*

Proof. We prove (i) the traces-refinement problem is in *PSPACE* and (ii) the traces-refinement problem is *PSPACE*-hard.

Firstly, we show (i) by providing a non-deterministic Turing machine that uses polynomial space to decide this problem. Let L_{sp} be the specification and $\mathcal{S}_I = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$, with induced LTS L_I and where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, the implementation supercombinator machine we are checking. We propose NTM $\textsc{Check}_T$ that runs on supercombinator machine $\mathcal{S}'$. $\mathcal{S}'$ combines the original implementation machine $\mathcal{S}$ with the specification L_{sp} in a way that an event performed by $\mathcal{S}$ has to be matched (synchronised) with L_{sp}. This machine is proposed so its induced LTS $L' = (S', \Sigma', \Delta', \hat{s}')$ captures the specification-implementation product space that is commonly explored in refinement checking.

$$\mathcal{S}' = (\langle L_1, \ldots, L_n, L_{sp} \rangle, \{((e_1, \ldots, e_n, a), a) \mid ((e_1, \ldots, e_n), a) \in \mathcal{R}\})$$

Then $\textsc{Check}_T$ is a slightly modified version of the machine $\textsc{Check}$ in Algorithm 2.1 that, instead of checking *blocked*(s), checks whether the currently-being-visited implementation state can perform an event that cannot be performed by the specification counterpart state. More formally, for state $s = (s_1 \ldots, s_n, s_{sp})$, it tests $\mathit{refines}_T(s) = \mathit{initials}_{L_I}((s_1, \ldots, s_n)) \not\supseteq \mathit{initials}_{L_{sp}}(s_{sp})$[1]. To check whether the refinement expression holds, we call machine $\textsc{Check}_T$ with input $(\mathcal{S}', \hat{s}', |S'|)$, where $|S'| = (\prod_{i \in \{1 \ldots n\}} |S_i|) \cdot |S_{sp}|$. If it accepts this input, the refinement expression does not hold, if it rejects, it does.

For reasons similar as the ones for $\textsc{Check}$ in Algorithm 2.1, our procedure both uses polynomial space and correctly asserts whether the refinement expression holds.

Secondly, we show (ii) by providing a polynomial-time reduction from the single-component traces-refinement problem to the traces-refinement problem. We reuse the machine $\mathcal{T}(L_{sp}, L_I)$ we propose for showing claim (ii) in the proof of Theorem 2.16 with CC slightly modified. We add an extra transition to the state *bad*; a self loop with fresh event err. Hence, we can translate the single-component traces-refinement problem involving L_{sp} and L_I into the traces-refinement problem involving L'_{sp} and $\mathcal{T}(L_{sp}, L_I)$, where L'_{sp} has a single state with self loops performing all events $\mathcal{T}(L_{sp}, L_I)$

[1] Given its simplicity and similarity to $\textsc{Check}$, we only informally describe $\textsc{Check}_T$.

can perform with the exception of err. For reasons similar to the ones presented in the proof to claim (ii) of Theorem 2.16, this refinement expression correctly asserts $L_{sp} \sqsubseteq_T L_I$ and it can be constructed in polynomial time on $|L_{sp}| + |L_I|$. $\square$

The stable-failures refinement problem makes use of an extended version of a LTS. When normalising a specification, refusals information about multiple LTS states may need to be combined into a single normalised state. Our LTS definition, however, can only (implicitly) capture the set of refusals $\{X \subseteq \mathcal{E} - initials(S)\}$. So, a more general structure is needed[2]. For this purpose FDR4 uses a *generalised-labelled-transition system* (GLTS) that annotates LTS states with a set of minimal acceptances, using function $Accs(s)$. A minimal acceptance Acc captures the set of refusals $\{X \subseteq \mathcal{E} - Acc\}$. Minimal acceptances are preferred over maximal refusals because they tend to provide a more compact representation for the same set of refusals.

Definition 2.20. A generalised labelled transition system $(S, \Sigma, \Delta, \hat{s}, Accs)$ is a 5-tuple, where $(S, \Sigma, \Delta, \hat{s})$ is a LTS and $Accs$ is a function from S to $\mathbb{P}(\mathbb{P}(\mathcal{E}))$ used to annotated states with sets of minimal acceptances. The predicate *ref* is defined differently for this structure. For $s \in S$, we have that $s\ ref\ X$ holds if and only if $\exists\, Acc : Accs(s) \bullet X \subseteq \mathcal{E} - Acc$. The size of a GLTS is given by the size of its underlying LTS plus the sum of minimal acceptances for all states.

We freely use a GLTS in a definition that deals with a LTS with the natural understanding that it uses the GLTS's underlying LTS structure. Moreover, we reinforce that the acceptances/failures information of a GLTS is not a property of the events offered/refused by the GLTS. This information is, instead, explicitly given by the minimal acceptances function $Accs$.

So, the failures-refinement problem is stated as follows.

Definition 2.21. Given a supercombinator machine $\mathcal{S}_I$ and a normalised specification GLTS L_{sp}, the failures-refinement problem asks whether $\mathcal{S}_I$'s induced LTS L_I refines L_{sp}, namely, whether $L_{sp} \sqsubseteq_F L_I$ holds.

As for traces, FDR4's refinement-checking engine normalises the specification and carries out the exploration of the product space of specification and implementation, looking for a specification-implementation pair of states (s_{sp}, s_I) such that s_I can perform an event that s_{sp} cannot or s_I can refuse to perform more events than s_{sp}

[2]Note this problem does not happen for the traces model. For that model, normalised states might have to store *initials* information about multiple states. Nevertheless, these multiple sets can be simply combined into a single set of initials.

(i.e. $initials(s_{sp}) \not\supseteq initials(s_I)$ or $\neg \exists Acc : Accs(s_{sp}) \bullet initials(s_I) \supseteq Acc$). If such a pair exists, the refinement expression does not hold. Otherwise, it does.

The problem of checking whether a failures-refinement expression holds is also *PSPACE*-complete. Hence, the lack of scalability of automatic verification techniques in tackling this problem.

Theorem 2.22. *The failures-refinement problem is* PSPACE-*complete.*

Proof. We prove (i) the failures-refinement problem is in *PSPACE* and (ii) the failures-refinement problem is *PSPACE*-hard.

To prove (i), we use the same argument we used to prove the corresponding claim for Theorem 2.19. The only modification is that we use machine $\textsc{Check}_F$ that is exactly as $\textsc{Check}_T$ with the exception that instead of checking $refines_T(s)$, it checks, for state $s = (s_1 \ldots, s_n, s_{sp})$, $refines_F(s) = initials_{L_I}((s_1, \ldots, s_n)) \not\supseteq initials_{L_{sp}}(s_{sp}) \vee \neg \exists Acc : Accs_{sp}(s_{sp}) \bullet initials_{L_I}((s_1, \ldots, s_n)) \supseteq Acc$, where $Accs_{sp}$ is the minimal-acceptances-annotation function for GLTS L_{sp}.

Secondly, we show (ii) by providing a polynomial-time reduction from the deadlock-freedom problem to the failures-refinement problem. Deadlock freedom can be checked by a standard refinement expression where $\mathcal{S}_I$ is the system being checked deadlock free and $L_{sp} = (\{s_0\}, \Sigma_I, \{(s_0, e, s_0) \mid e \in \Sigma_I\}, s_0, Accs(s_0) = \{\{e\} \mid e \in \Sigma_I\})$ is the deadlock-freedom specification (Σ_I is the alphabet of $\mathcal{S}_I$). This refinement expression correctly asserts whether $\mathcal{S}_I$ is deadlock free and it can be constructed in polynomial time on the size of $\mathcal{S}_I$. □

2.2.3 Taming the state-space explosion problem

In this section, we discuss partial order reduction and compression techniques. These methods are implemented in FDR4 to help tame the state-space explosion problem in analysing induced LTSs.

2.2.3.1 Partial order reduction

Partial order reduction is a widely known technique that can soundly reduce the state space of a system's induced LTS. An implementation of this technique within FDR4 has been proposed in [GRHRW15]. Most concurrency models represent independent transitions by *interleaving*, namely, they allow independent transitions to happen in any given order. Two transitions are said to be independent when taking one of them does not preclude the other from being enabled. One example of such transitions are component transitions that do not require synchronisation. If some independent

transitions are enabled in a given state, then all the paths interleaving these transitions are valid. So, for n independent transitions, there could be $n!$ possible sequences and 2^n states to be checked.

However, even though all these combinations are possible, for a considerable class of properties the examination of only one ordering of transitions is sufficient for yielding whether the property is valid. Deadlock freedom is an example of such a property. So, rather than exploring 2^n states, partial order reduction allows a given property to be verified by only visiting $n + 1$ states, namely, the states along a single path involving independent transitions. For a detailed presentation and evaluation of such a techniques, one should refer to [GW93, GRHRW15].

For these techniques, there is an intrinsic trade-off between the optimality of the reduction and the cost to calculate and perform it. Usually, the more optimal the reduction is intended to be, the bigger is the overhead to calculate it and to carry it out. Furthermore, the success of this method is closely tied to how many independent transitions a system has; if the reduction in the state space does not compensate the cost of carrying out this reduction, this technique can be even worse than the simple state-space exploration [GRHRW15].

2.2.3.2 Compression

In [RGG+95], Roscoe *et al.* propose some techniques that, broadly speaking, try to reduce a given LTS by merging similar states. For instance, one of these techniques uses the traditional notion of *strong bisimilarity* to merge states. This work also advocates for the systematic use of these techniques in the process of constructing the system to be analysed from its components; this approach is called *hierarchical compression.* This systematic approach consists of a stepwise construction where each step amounts to the parallel combination of some components and the compression of this newly created subsystem. Hopefully, the successive compressions keep the state space of subsystems reasonably small. As an example of how effective this technique can be, a systematic construction of the dining philosophers, proposed in the same work, enables the finding of a deadlock for a system with 10^{1000} philosophers in 15 minutes.

Hierarchical compression also presents some shortcomings. Firstly, a compression technique is not guaranteed to reduce the state space of a system, for similar states might not exist. Secondly, the effectiveness of this stepwise approach might be considerably affected by the ordering or the granularity of the compositions that have to be made. A poor choice of ordering or granularity might cause a blow up to the

state space for some intermediate subsystem. Thus, one must be very skilful and experienced to *manually* devise an effective construction strategy.

2.3 SAT solving

A SAT solver decides whether a given *propositional formula* is satisfiable, namely, it tries to establish whether there exists an assignment for the boolean variables in this formula that makes the formula evaluate to *true*. If the formula is satisfiable, the solver also returns a satisfying assignment, which is usually called a *model*. Given a model $\mathcal{A}$ and a boolean variable x_i, $\mathcal{A}(x_i)$ gives the boolean value assigned to x_i in $\mathcal{A}$.

Let $x_1, \ldots, x_n$ be boolean variables. A well-formed propositional formula $\mathcal{F}$ is constructed inductively using the following grammar:

$$\mathcal{F} ::= x_i \mid \neg\mathcal{F} \mid (\mathcal{F}) \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \mathcal{F}_1 \vee \mathcal{F}_2$$

We use $\bigwedge_i \mathcal{F}_i$ ($\bigvee_i \mathcal{F}_i$) as a shorthand for the conjunction (disjunction) of formulas F_i.

Most solvers take as an input a propositional formula in *Conjunctive Normal Form* (CNF). A formula is in CNF if it is a conjunction of *clauses*. A clause is a disjunction of *literals*, where a literal is a boolean variable or its negation. Any propositional formula can be translated, in linear time, into an *equisatisfiable* CNF formula using *Tseitin's encoding* [Tse68]. Two formulas are equisatisfiable if one is satisfiable if and only if the other is. In this encoding, a sub-formula $\mathcal{F}$ is abstracted by a fresh boolean variable $x_\mathcal{F}$ (i.e. $x_\mathcal{F} \Leftrightarrow \mathcal{F}$); this abstraction is encoded via a CNF formula. So, a non-CNF (sub-)formula can be replaced by its abstracting variable.

For examples, let $x_\mathcal{F}$ be a variable abstracting formula F.

- $\mathcal{F}' = \neg\mathcal{F}$ translates to $(x_{\mathcal{F}'} \vee x_\mathcal{F}) \wedge (\neg x_{\mathcal{F}'} \vee \neg x_\mathcal{F})$
- $\mathcal{F}' = \mathcal{F}_1 \vee \mathcal{F}_2$ translates to $(\neg x_{\mathcal{F}'} \vee x_{\mathcal{F}_1} \vee x_{\mathcal{F}_2}) \wedge (x_{\mathcal{F}'} \vee \neg x_{\mathcal{F}_1}) \wedge (x_{\mathcal{F}'} \vee \neg x_{\mathcal{F}_2})$
- $\mathcal{F}' = \mathcal{F}_1 \wedge \mathcal{F}_2$ translates to $(x_{\mathcal{F}'} \vee \neg x_{\mathcal{F}_1} \vee \neg x_{\mathcal{F}_2}) \wedge (\neg x_{\mathcal{F}'} \vee x_{\mathcal{F}_1}) \wedge (\neg x_{\mathcal{F}'} \vee x_{\mathcal{F}_2})$

The boolean satisfiability problem is *NP*-complete [Coo71]. So, currently, only procedures that take exponential time in worst case are known, and assuming $P \neq NP$, there cannot be a decision procedure that generally solves this problem in polynomial time. Modern SAT solvers, however, can decide many practical instances of this problem very efficiently. This efficiency is due to a combination of clever techniques that prune the exponentially-large search space.

2.3.1 The DPLL framework

Modern SAT solvers are based on the DPLL (Davis Putnam Logemann Loveland) framework [DP60, DLL62]. It implements a backtracking-based search that tries to systematically construct a satisfying assignment for a given formula. If no satisfying assignment can be constructed, the formula is unsatisfiable. Roughly speaking, it starts with the empty assignment, where no variables are assigned, and progressively extends this assignment by choosing an unassigned boolean variable and deciding on its value, then it propagates the implications of this decision, and it backtracks if propagation leads to a *conflict*. Given a partial assignment, a clause is *conflicting* (or, a conflict) when all its literals (and the clause itself) evaluate to false, *satisfied* when one of its literals evaluates to true, *unit* when it is not satisfied and all but one of its literals has yet to have its variable assigned to a value, or *unresolved* otherwise, i.e. a non-satisfied clause with more than one literal unassigned. We use the formula in Example 2.3 as a running example to illustrate some notions introduced in this section. Considering this formula and assignment $\{x_7 = false, x_8 = true\}$, clauses c_2, c_3, c_5, c_7 are satisfied, c_6 is conflicting, and c_1, c_4 are unit.

Example 2.3. Boolean formula $\mathcal{F}_{ex}$ is as follows.

Algorithm 2.2 Davis-Putnam-Loveland-Logemann (DPLL) algorithm takes a propositional formula in CNF format and outputs SAT (satisfiable) or UNSAT (unsatisfiable).

```
1: function Solve(F)
2:     while true do
3:         propagate()
4:         if no conflict found then
5:             if all variables assigned then
6:                 return SAT
7:             else
8:                 decide()
9:             end if
10:        else
11:            level = analyse_conflict()
12:            if level = −1 then
13:                return UNSAT
14:            else
15:                backtrack(level)
16:            end if
17:        end if
18:    end while
19: end function
```

$$
\begin{aligned}
&c_1 = x_1\\
&c_2 = \neg x_1 \lor x_2 \lor x_7 \lor x_8\\
&c_3 = x_2 \lor \neg x_7 \lor \neg x_8\\
&c_4 = \neg x_2 \lor x_7 \lor \neg x_8\\
&c_5 = \neg x_2 \lor \neg x_7 \lor x_8\\
&c_6 = x_7 \lor \neg x_8\\
&c_7 = \neg x_7 \lor x_8
\end{aligned}
$$

■

We present a sketch of the DPLL framework in Algorithm 2.2 and briefly describe the functions it uses as follows. Function *propagate*() infers new variable assignments based on the current partial assignment. Note a unit clause can only be satisfied if the current assignment is extended so the one unassigned literal evaluates to true. We call this inference mechanism the *unit-clause rule*. So, *propagate*() tracks unit clauses and uses the unit-clause rule to extend the current assignment until either a conflict is found or no more unit clauses exist. This mechanism is generally known as *boolean constraint propagation*. We point out that this propagation can even happen when considering the empty assignment, as it does, for instance, in Example 2.3. This is why the function *propagate*() is called right at the start of this algorithm.

After this propagation is carried out, the next step taken depends on whether a conflict has been found. If no conflict is detected, this framework continues its search by assigning a value to an unassigned variable. Function *decide*() is responsible for choosing this variable-value pair to extend the current assignment with. The assignment space can be seen as decision tree where nodes are variables and each outgoing edge represents a valuation for the associated node/variable. Broadly speaking, the DPLL framework can be understood as looking for a satisfying assignment by implicitly traversing this tree. So, each time *decide*() is called a new branch in this decision tree is being explored. For each decision/branching, *decision level* $dl + 1$ is created. The variable assignment picked by *decide*() and all propagated assignments are associated with this level. By convention, initially $dl = 0$ and this level is not associated to any decision. We use $x = v@dl$ to state that the variable assignment $x = v$ is associated with decision level dl. Also, $\underline{x = v@dl}$ means that this is the decision assignment for level dl. As decision levels keep track of branching points in the algorithm, they play an integral part in the backtracking process. Modern solvers choose a variable to assign based on a heuristic that prioritises variables more frequently involved in conflicts. We detail the heuristic commonly used by solvers and how it can help with the solving process in Section 2.3.2.

If propagation leads to a conflict, function $analyse_conflict()$ analyses which decisions have led to the conflict. This analysis generates a new clause which prevents this conflict from arising again and returns the highest decision level that does not propagate the conflict. If this analysis return decision level -1, it means that no decision can lead to a satisfying assignment and the formula must be unsatisfiable. Otherwise, this framework backtracks to the returned decision level, by calling $backtrack(level)$. This function undoes all the variable assignments for decision levels $dl > level$. By backtracking to the second highest decision level considering the decision variables causing a conflict, it undoes the last decision that implies the conflict. This backtracking and the new *learned* clause trigger the propagation of the opposite assignment for this last decision variable. We detail this conflict-based clause-learning process and its impact on the DPLL framework in Section 2.3.3.

In Figure 2.6, we illustrate how this algorithm works by describing a partial execution for our running example. For this run, we disregard decision heuristics and choose variable-value pairs which, we believe, create a more complete picture of how this DPLL algorithm works. We make the following remarks about this execution. Firstly, Step 3 is omitted as it consists of a pointless propagation call, namely, a call to $propagate()$ with no unit clauses. Secondly, Step 6 results in learned clause $c_8 = x_2 \vee x_7$, as the analysis of the conflict in c_6 reveals that decisions $x_2 = false$ and $x_7 = false$ are its cause. Note how this clause prevents these conflicting decisions from occurring again; making one of these decisions would cause this learned clause to propagate the opposite assignment for the other variable. Finally, note how, in Step 7, backtracking to level 1 causes c_8, the learned clause, to become unit, which in turn leads to the propagation of $x_7 = true$ in Step 8. This propagation corresponds exactly to flipping the decision $x_7 = false$ which caused conflict c_6 in Step 5.

Modern SAT solvers are very conflict centric; both the decision heuristic they normally employ and their backtracking strategy are conflict-based. This emphasis on conflicts is arguably the driving force behind the impressive efficiency of SAT solvers in checking satisfiability. The intuition behind this emphasis is that conflicts provide valuable information that can be used to prune the search space.

2.3.2 Decision heuristic

An integral part of solvers' efficiency comes from their decision heuristics, namely, the method they use to select which variable, value to assign next. Most solvers use a variation of *Variable State Independent Decaying Sum* (VSIDS) as a heuristic to make this choice. This heuristic scores each literal based on the number of clauses in which

Step 0: starting point
Current assignment: $\{\}$
Current decision level: 0
unresolved: $\{c_2, c_3, c_4, c_5, c_6, c_7\}$
unit: $\{c_1\}$
satisfied: $\{\}$
conflict: $\{\}$

Step 1: propagate()
Current assignment: $\{x_1 = true@0\}$
Current decision level: 0
unresolved: $\{c_2, c_3, c_4, c_5, c_6, c_7\}$
unit: $\{\}$
satisfied: $\{c_1\}$
conflict: $\{\}$

Step 2: decide()
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}\}$
Current decision level: 1
unresolved: $\{c_2, c_3, c_6, c_7\}$
unit: $\{\}$
satisfied: $\{c_1, c_4, c_5\}$
conflict: $\{\}$

Step 4: decide()
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}, \underline{x_7 = false@2}\}$
Current decision level: 2
unresolved: $\{\}$
unit: $\{c_2, c_6\}$
satisfied: $\{c_1, c_3, c_4, c_5, c_7\}$
conflict: $\{\}$

Step 5: propagate()
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}, \underline{x_7 = false@2}, x_8 = true@2\}$
Current decision level: 2
unresolved: $\{\}$
unit: $\{\}$
satisfied: $\{c_1, c_2, c_3, c_4, c_5, c_7\}$
conflict: $\{c_6\}$

Step 6: analyse_conflict() = 1
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}, \underline{x_7 = false@2}, x_8 = true@2\}$
Current decision level: 2
unresolved: $\{\}$
unit: $\{\}$
satisfied: $\{c_1, c_2, c_3, c_4, c_5, c_7\}$
conflict: $\{c_6\}$
learned: $c_8 = x_2 \vee x_7$

Step 7: backtrack(1)
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}\}$
Current decision level: 1
unresolved: $\{c_2, c_3, c_6, c_7\}$
unit: $\{c_8\}$
satisfied: $\{c_1, c_4, c_5\}$
conflict: $\{\}$

Step 8: propagate()
Current assignment: $\{x_1 = true@0, \underline{x_2 = false@1}, x_7 = true@1, x_8 = false@1\}$
Current decision level: 1
unresolved: $\{\}$
unit: $\{\}$
satisfied: $\{c_1, c_2, c_3, c_4, c_5, c_6, c_8\}$
conflict: $\{c_7\}$

Figure 2.6: Possible partial DPLL-framework run for Running Example 2.3.

it appears, and it periodically divides all scores by 2^3. So, *decide*() picks whichever unassigned literal has the highest score to assign. By assigning a literal, we mean that its variable is assigned to the value that makes the literal *true*.

This heuristic aims to prioritise literals/variables that have frequently and recently caused conflicts. The clause learning feature increases literals' scores every time they

[3]This constant is normally a parameter of solvers.

cause a conflict, and the periodic division of scores makes recent conflicts more relevant than older ones. The intuition behind it is that variables at the cause of conflicts are more directly related to the satisfiability of a formula. By prioritising recent conflicts, it hopes to pick variable-value pairs that are more intrinsically related to the part of the assignment space the algorithm is currently at.

We illustrate how this heuristic can prune the search space with the help of formula $\mathcal{F}_{ex}$ in Example 2.3. Let us assume that our algorithm is trying to solve formula $\mathcal{F}_{ex} \wedge \mathcal{F}'$, which involves variables $x_1, \ldots, x_7$. $\mathcal{F}_{ex}$ specifies that $x_2 \Leftrightarrow x_7 = x_8$ and $x_7 = x_8$. So, while $x_2 = false$, no assignment can satisfy our formula. Assuming that $x_2 = false$ is decided first and x_7 immediately after, these values alone would lead to a conflict by propagation without the need to assign values for variables $x_3, \ldots, x_6$. On the other hand, if variables $x_3, \ldots, x_6$ are assigned in between $x_2 = false$ and x_7, the solver would need to (possibly) waste time assigning values for $x_3, \ldots, x_6$ only to realise that x_7 is the relevant variable for this part of the search. Therefore, identifying x_7 as a relevant variable and picking it early can substantially reduce the search space to be explored.

2.3.3 Conflict-driven backtracking

Another key feature in making solvers efficient is the ability to jump over many decision levels, which is commonly referred to as *non-chronological backtracking*. A conflict can give valuable information about the decisions (and decision levels) that have caused it. By tracking decisions and their implications, solvers are able to deduce the decisions implicated in a conflict. This tracking is kept by an *implication graph*. This digraph has a node for each variable-value pair in the current assignment, except for variable assignments made in level 0, and a special conflict node k. There is an edge from x_i to x_j in this graph if the value of x_j was implied/propagated thanks to the value of x_i, and there is an edge from x_i to k if x_i is a variable in the conflict (clause) found. So, in particular, decision variables have no incoming edges in this graph. For instance, in Figure 2.3.3, we show the implication graph at Step 6 of the execution run in Figure 2.6. We use literals to abbreviate assignments, namely, $\neg x_i$ means $x_i = false$ whereas x_i means $x_i = true$. Also, we annotate the edges with the clause that caused the propagation.

Roughly speaking, the function $analyse_conflict()$ traverses this graph in a backward fashion from k and find nodes without incoming edges, namely, decision assignments. These nodes correspond to the decisions that have effectively played a part in making the current assignment lead to a conflict. Based on the set of decision

nodes found, it calculates a decision level to backtrack to and a clause to learn. The backtracking level is the second highest level associated with these nodes[4], and the learned clause is composed of the negation of these literals[5]. For instance, for the graph in Figure 2.3.3, this function finds nodes $x_2 = false$ and $x_7 = false$. So, it returns decision level 1 and it learns/generates clause $x_2 \lor x_7$.

We illustrate how this conflict-driven-backtracking feature can prune the search space with the help of formula $\mathcal{F}_{ex}$ in Example 2.3. Let us assume that our algorithm is trying to solve formula $\mathcal{F}_{ex} \land \mathcal{F}'$, which involves variables $x_1, \ldots, x_7$. Assuming that decisions are made based on variable index and that this algorithm finds a conflict for assignment $\{\underline{x_2 = false@1}, \underline{x_3 = false@2}, \ldots, \underline{x_6 = false@5}, \underline{x_7 = false@6}, x_8 = true\}$. If the values for variables $x_3, \ldots, x_6$ do not contribute to this conflict, the algorithm would have constructed an implication graph for which the sub-graph containing nodes that are backward reachable from k is shown in Figure 2.3.3. By analysing this graph, *analyse_conflict*() would detect decision nodes $x_2 = false$ and $x_7 = false$ as causes of the conflict. So, it would return level 1 and learn clause $x_2 \lor x_7$. Backtracking to level 1 makes the learned clause become unit and so $x_7 = true$ is propagated. We point out that this backtracking (from level 6 to level 1) saves the time of assigning different values for variables $x_3, \ldots, x_6$ as they are not the source of this conflict. This effort would not be saved if a chronological approach was in place; such an approach would backtrack to level 5. Therefore, by identifying and performing a non-chronological backtracking, this algorithm can substantially prune the search space.

[4] If no such nodes are found, it returns level -1. If a single such node is found, it returns level 0.

[5] There are some optimisations for this methods using unique implication points that are beyond the scope of this thesis. For more details see [KS08].

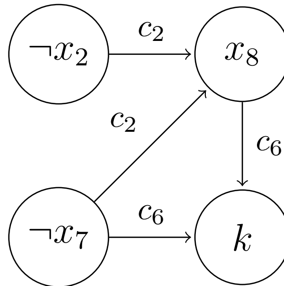

Figure 2.7: Implication graph at Step 6 of execution in Figure 2.6.

2.3.4 Encoding compactness and quality

How a problem is encoded also plays a big part in how efficiently it can be solved. The techniques introduced later on this thesis involve encoding a problem into a boolean formula and checking it using a solver. In devising encodings for these techniques, we often came across a trade-off between size and quality of our translated formula. By quality, we mean that the translation should preserve as many *implications* between literals in the translated formula as there are between the elements they represent in the original problem. When a translation preserves these implications, we call it *arc-consistent*[6]. An intuitive explanation of arc-consistency is given in [ES06]: "whenever an assignment can be propagated on the original constraint, the SAT solver's unit propagation, operating on the translated formula, should find that assignment too". This means, in particular, that an assignment that implies a conflict in the original problem also implies a conflict for the translated formula. This preservation of implications can save the solver a lot of effort in checking a problem. For instance, when the solver decides on a partial assignment that leads the original problem to a conflict, instead of deciding on extra unassigned variables to then realise a satisfying assignment is not possible, it can directly deduce so.

We illustrate this trade-off with an example: we want to encode into a SAT problem whether, for a given binary relation on V, there exists *no* reflexive pair in its transitive closure, namely, whether a given directed graph is acyclic. So, it returns SAT if the graph is acyclic and $UNSAT$ if it has a cycle.

One way to encode this is by associating each node $a \in V$ in this digraph to a integer i_a and making sure that for each edge (a, b) in this graph, we encode that $i_a > i_b$. So, an assignment of integers to nodes exists if and only if the underlying graph is cycle-free. We call this encoding *integer-based*. To make this encoding work, integer variables i_a need to be in the range $\{1 \dots |V|\}$; we need enough values to, in the worst case scenario, be able to assign a different integer to each node $a \in V$.

In terms of size, this encoding is rather compact. It needs $\mathcal{O}(|V|\ log|V|)$ variables, each integer variable i_a for $a \in V$ is encoded (in binary) by $log|V|$ boolean variables, and $\mathcal{O}(|V|^2)$ clauses, to encode the edges $i_a > i_b$. As for quality, this encoding is not optimal, as it is far from being arc-consistent. For instance, let us assume that the underlying graph has a cycle. This encoding would need to try on different values for integer variables i_a until it finds out that no satisfying assignment can be found.

[6]For a more formal and detailed account see [ES06].

Another way to encode this problem is by capturing the transitive closure of this relation and asking whether its has a reflexive pair. To capture the pairs in the transitive closure, it associates each pair $(a, b) \in V^2$ with a boolean variable $e_{a,b}$. It encodes that pair/edge (a, b) is a member of the original relation/digraph by creating unit clauses $e_{a,b}$. Then, it creates some axioms to encode transitivity, namely, for each triple of transitions $e_{a,b}$, $e_{b,c}$ and $e_{a,c}$, it encodes the axiom: $e_{a,b} \wedge e_{b,c} \Rightarrow e_{a,c}$. Finally, it encodes that a reflexive pair cannot occur by adding $\neg e_{a,a}$ for every $a \in V$.

In terms of size, this encoding is not very compact. It requires $\mathcal{O}(|V|^2)$ variables, one for each edge/pair (a, b) in the digraph/relation, and $\mathcal{O}(|V|^3)$ clauses, to encode the pairs in the relation, transitiveness axioms and that reflexive pairs cannot be derived. In terms of quality, this encoding is arc-consistent. With implication alone, it can derive whether the graph is acyclic or not. The encoding of the original pairs in the relation and the axioms for transitiveness cause the solver to construct the transitive close of the relation through implication (boolean propagation) and this leads to establishing whether the underlying digraph is acyclic.

This example shows the sort of trade-off we experienced in proposing SAT encodings for our frameworks. It seems that it is often the case that quality and size are two opposing forces when encoding a problem into SAT. As we have demonstrated for our acyclic-graph problem: on one hand, a compact encoding can be created with poor implication quality, on the other, optimal implication can be achieved by a not-so-compact translation. Note that it does not even appear to be a trade-off to be contemplated at all; the difference in size for these encodings is rather small whereas the larger encoding is substantially better in terms of implications preservation. So, there should be no discussing in picking the second encoding over the first one. In practice, however, the decision heuristic used by solvers seems to be very efficient/accurate in picking the "right" values for the integer variables in the first encoding. Moreover, a cubic growth in the number of clauses is often sufficient to completely hinder the efficiency of solvers. We were not the only ones to come across these practical points, other works have faced the same issues [GJR14].

2.4 SMT solving

Satisfiability modulo theories is the name that is generally given to the problem of determining the satisfiability of a *quantifier-free formula* with respect to some *theory* [BT18]. A *first-order theory* T, or just *theory* for short, over the set of non-logical symbols S (called its *signature*) combines first-order logic with a fixed

interpretation for elements in S. This interpretation is often given by a set of axioms. So, a well-formed S-formula (i.e., a first-order formula for which non-logical symbols come from S) is T-satisfiable (or T-consistent) if and only if it is satisfiable considering the interpretation given by T for symbols in S. First-order logic is meant to be a generic framework where non-logical symbols can have arbitrary meaning. In practice, however, one is often interested in a specific interpretation for a non-logical symbol. Therefore, the interest in studying first-order theories.

Example 2.4. Equality logic is a theory over the signature $S = \{=\}$, where $=$ is a binary predicate. The following formula is well-formed:

$$(a = b) \wedge (b = c) \wedge \neg(a = c)$$

This theory also constrains the meaning of "=" to the equality predicate. This interpretation is captured by the usual three axioms:

$$\begin{array}{ll} x = x & reflexivity \\ x = y \Longrightarrow y = x & symmetry \\ x = y \wedge y = z \Longrightarrow x = z & transitivity \end{array}$$

Our formula is unsatisfiable for this theory as transitivity implies that $(a = c)$. ■

A *SMT solver* is a decision procedure that solves such satisfiability problems. Most solvers consider this problem for a combination of theories. So, given a S-formula and theories $T_1, \ldots, T_n$ over S, a SMT solver answers whether this formula is satisfiable (SAT) or not (UNSAT). In the SAT case, it provides a model/satisfying assignment $\mathcal{A}$ for the variables in the formula. $\mathcal{A}(x)$ gives the value assigned to x in model $\mathcal{A}$.

2.4.1 Linear integer arithmetic

A theory that is commonly handled by SMT solvers and that is used in this work is linear integer arithmetic. Aside from traditional propositional boolean variables, this theory allows for arithmetic atomic propositions.

Definition 2.23. An arithmetic atomic proposition is constructed using the following grammar, where x is some integer variable and c some integer value.

$$\begin{array}{l} atom ::= sum\ op\ sum \\ op ::= = \mid \leq \mid < \\ sum ::= term \mid sum + term \\ term ::= x \mid c \mid c \cdot x \end{array}$$

This theory interprets arithmetic atoms in the usual way. Next, we present a formula that illustrate the use of linear arithmetic atoms.

Example 2.5. Let x_1, x_2 be boolean variables, and x_3, x_4 be integer variables.

$$(x_1 \vee x_2) \wedge ((\neg x_1 \vee (x_3 + 4x_4 = 3 \wedge 1 \leq x_4)) \vee (\neg x_2 \vee (0 < x_3 - x_4 \wedge x_3 \leq x_4)))$$

■

This formula is not satisfiable according to this theory. Note that either x_1 or x_2 must be set to *true*, and so, in each case, a couple of inconsistent arithmetic atoms must be satisfied to satisfy the formula. Also, we point out that this formula is not in conjunctive normal form. SMT solvers do not usually require their input formula to be in CNF, but they implement a pre-processing procedure to convert a formula into CNF using Tseitin's encoding.

2.4.2 The DPLL(T) framework

Modern SMT solvers leverage the advances on SAT solvers to efficiently check for satisfiability. These solvers, usually, implement satisfiability checking in one of two ways: *eagerly* or *lazily*. Eager solvers carry out a one-step translation of a theory-specific formula into an equisatisfiable propositional formula. Note that this generated formula has to include theory-specific consequences encoded in a propositional form to ensure the generated formula and the original one are equisatisfiable. This generated formula is fed to a SAT solver so it can check whether it is satisfiable. Such approaches also maintain a mapping between elements in the original formula and elements in the generated formula so that a model for the generated formula, created by the SAT solver, can be "lifted" to a model for the original formula. Naive translations into SAT usually result in poor checking performance. So, these implementations often rely on theory-specific optimisations, making eager approaches very theory specific. The sort of translation implemented by such approaches is very similar to the one discussed in Section 2.3.4. Therefore, we point out that eager approaches have to deal with a similar trade-off between quality and size of encodings. One example of this sort of solver is UCLID [LS04].

Most of the modern SMT solvers are implemented using the lazy approach, which involves tightly coupling a DPLL algorithm with a *theory solver*. A theory solver is a decision procedure that solves the satisfiability problem for the *conjunctive fragment* (i.e. the set of quantifier-free formulas assembled using only conjunctions) of a corresponding theory. For instance, theory solvers for linear integer arithmetic are mostly based on the *branch and bound algorithm* for solving conjunctions of linear constraints over integers [KS08]. This combination DPLL, theory-solver can be

Algorithm 2.3 DPLL(T) algorithm takes a theory-specific formula $\mathcal{F}$ and outputs SAT (satisfiable) or UNSAT (unsatisfiable).

```
 1: function Solve(F)
 2:     F^p = preprocess(F)
 3:     while true do
 4:         propagate()
 5:         if no conflict found then
 6:             if T-deduce(T_∧(A^p)) then
 7:                 if all variables assigned then
 8:                     return SAT
 9:                 else
10:                     decide()
11:                 end if
12:             end if
13:         else
14:             level = analyse_conflict()
15:             if level = −1 then
16:                 return UNSAT
17:             else
18:                 backtrack(level)
19:             end if
20:         end if
21:     end while
22: end function
```

understood as lazily constructing a translation of the original theory-specific formula into a propositional equisatisfiable one. This framework is generally known as DPLL(T) framework, where T is a background theory. Modern solvers can check a formula involving a combination of different theories, however, for our brief explanation on how DPLL(T) works, we only consider a single theory. For an account on how combination of theories can be handled see [BBH$^+$09], Chapter 12, or [KS08], Chapter 10.

Lazy approaches have the advantage of implementing theory-specific solvers that can use whatever specialised algorithms and data structures are best for the theory in question, which typically leads to better performance. Eager approaches, on the other hand, have the advantage that they rely on a translation that imposes upfront all theory-specific constraints on the SAT solvers search space, which can potentially help the solver to check the input formula more quickly. So, its viability depends on the ability of modern SAT solvers to quickly process relevant theory-specific information encoded into large SAT formulas.

We present a sketch of the DPLL(T) framework in Algorithm 2.3. Lazy frameworks

work on two levels: a propositional level, where simple DPLL features are used, and a theory level, where a theory solver is invoked. To build on the remarkable efficiency of SAT solvers, SMT checkers work on a *propositional skeleton* of the input formula. For a formula $\mathcal{F}$, its propositional skeleton is given by $\mathcal{F}^p$ and consists of replacing theory-specific literals by propositional boolean variables. Moreover, to work simultaneously and consistently on these two levels, SMT checkers also maintain a mapping between boolean variables in $\mathcal{F}^p$ and the theory-specific literals they correspond to in the original formula $\mathcal{F}$. The function $preprocess(\mathcal{F})$ is responsible for creating both this propositional skeleton and mapping, converting the formula into conjunctive normal form, and finally, applying theory-specific simplifications to, possibly, reduce the complexity of solving this formula.

Example 2.6. Let x_1 be a boolean variable and x_2, x_3 be integer variables.

$$\mathcal{F} = x_1 \wedge (\neg x_1 \vee x_2 = x_3) \wedge (x_1 \vee \neg(x_2 - x_3 = 1))$$

■

For instance, the propositional skeleton of the formula in Example 2.6 is given as follows; boolean variables x_4 and x_5 represent (and are mapped to) arithmetic literals $x_2 = x_3$ and $\neg(x_2 - x_3 = 1)$, respectively.

$$\mathcal{F}^p = x_1 \wedge (\neg x_1 \vee x_4) \wedge (x_1 \vee x_5) \tag{2.8}$$

This framework works analogously to DPLL for SAT checking with the exception for a call to $T\text{-}deduce(T_\wedge(\mathcal{A}))$ in line 6; this function call represents exactly the invocation of a theory solver. Roughly speaking, it iteratively uses functions $propagate()$, $decide()$, $analysed_conflict()$, $backtrack()$ to find a partial assignment $\mathcal{A}^p$ satisfying the propositional skeleton of the input formula, which is then checked for consistency with background theory by $T\text{-}deduce(T_\wedge(\mathcal{A}^p))$. $T_\wedge(\mathcal{A}^p)$ creates a conjunction of theory-specific literals. For each boolean variable x assigned in $\mathcal{A}^p$ and which corresponds to a theory-specific literal l in the original formula, it conjoins literal $\neg l$ if $x = false$ in $\mathcal{A}^p$ or l if $x = true$. For instance, let $\mathcal{A}^p = \{x_1 = true, x_4 = true, x_5 = false\}$ be the current partial assignment being considered for formula in Example 2.6 and its propositional skeleton in Equation (2.8), $T_\wedge(\mathcal{A}^p)$ would be given by $(x_2 = x_3) \wedge (x_2 - x_3 = -1)$. Theory solvers are usually designed to check for consistency (satisfiability) regarding the conjunctive fragment of a theory, hence the need for function $T_\wedge(\mathcal{A}^p)$.

Theory solvers not only check for satisfiability but they implement two important features: *theory propagation* and *lemma generation*. Theory propagation complements boolean constraint propagation implemented in DPLL by performing theory-specific

deductions. A boolean variable corresponding to a theory-specific literal is assigned to a value, if this valuation is the only possibility that makes the current partial assignment theory consistent. For instance, let $\mathcal{A}^p = \{x_1 = true, x_4 = true\}$ be the current partial assignment for Example 2.6, from this assignment theory propagation could immediately deduce that $x_5 = true$. This assignment is implied by the fact that if $x_2 = x_3$ (thanks to $x_4 = true$) then $\neg(x_2 - x_3 = -1)$ must be true (i.e. $x_5 = true$), otherwise it would lead to a theory inconsistency.

Lemma generation can be seen as a sort of conflict analysis on the theory level of SMT solvers. If *T-deduce*$(T_\wedge(\mathcal{A}^p))$ concludes that the conjunction $T_\wedge(\mathcal{A}^p)$ is inconsistent, it analyses the causes for this inconsistency and learns a blocking clause that prevents the same inconsistency from happening again. This process involves finding some literals $l_1, \ldots, l_m$ in $T_\wedge(\mathcal{A}^p)$ such that $l_1 \wedge \ldots \wedge l_m$ leads to an inconsistency, so the solver can learn (i.e. add to the propositional skeleton of the input formula) the negation of this conjunction. For instance, let $\mathcal{A}^p = \{x_1 = true, x_4 = true, x_5 = false\}$ be the current partial assignment for formula in Example 2.6 with propositional skeleton in Equation (2.8), this assignment leads to an inconsistency as $T_\wedge(\mathcal{A}^p)$ is $(x_2 = x_3) \wedge (x_2 - x_3 = -1)$. Both these literals are responsible for this inconsistency, so the SMT solver learns $\neg((x_2 = x_3) \wedge (x_2 - x_3 = -1))$, which is added to the propositional skeleton in boolean form: $\neg(x_4 \wedge (\neg x_5))$.

The call to *T-deduce*$(T_\wedge(\mathcal{A}^p))$ returns *true* if the theory solver neither carried out some theory propagation nor some lemma generation, and *false* otherwise. If it returns *false*, it means that the theory solver has carried out some change to the propositional skeleton or to the current partial assignment that might lead to some boolean propagation or a conflict, so our SMT solver goes back to operate on the boolean level by calling *propagate*(). On the other hand, if it returns *true*, the theory solver has found that the partial assignment is theory consistent and there is no boolean or theory propagation to be carried out at this point.

Finally, once a complete and theory-consistent satisfying assignment is found for the skeleton, a model $\mathcal{A}$ assigning consistent values for the theory-specific variables of the original input formula is generated. This feature is called *model generation*.

2.5 Summary

This chapter has presented some concepts, problems and tools upon which the work in this thesis is based.

We have presented the concept of supercombinator machines as a means to precisely capture the behaviour of concurrent and distributed systems. This is the formalism upon which our work is based. Even though this notation has been proposed in the context of CSP, it is flexible enough to capture systems described in other formalisms. To illustrate how a translation into this formalism can be achieved, we have presented how it captures systems described in a simplified version of the CSP language.

Based on this formalism, we have introduced three problems investigated in this thesis: deadlock-freedom checking, traces refinement checking and stable-failures refinement checking. These problems are all *PSPACE*-complete and, therefore, believed to be very hard to solve. This thesis investigates approximate frameworks to more efficiently tackle these problems.

In this thesis, we propose a number of approximate verification frameworks that rely on SAT/SMT solvers to check the hard problems they tackle. Hence, in this chapter, we briefly introduced how these solvers work and the conflict-centric heuristics that help them solve hard problems efficiently.

Chapter 3

Local analysis

3.1 Introduction

In many cases, a property emerges from the behaviour of small subsystems as opposed to the system's global behaviour. For instance, a pair of components in the system might always be able to communicate, so the behaviour of this pair alone can ensure that the system is deadlock free. In these cases, verification frameworks could benefit from employing *local analysis* to examine these subsystems and capture *local invariants*. A local invariant approximates the behaviour of the system based on the behaviour of one of its subsystems. It is meant to show that some system behaviours are invalid because the components in this subsystem cannot cooperate to perform them. The behaviours that cannot be shown invalid in this way are conservatively assumed to be valid. In this chapter, we propose the notion of *subsystem reachability* as a device to implement local analysis and capture local invariants. It over-approximates system reachability by showing that, for a chosen subsystem, some system states are not reachable because the components in this subsystem cannot cooperate to reach them.

We also use the notion of subsystem reachability to systematically create a family of reachability over-approximations. For $k \geq 1$, *k-reachability* proposes an approximation that combines reachability for some subsystems of size up to k. This systematisation is an attempt to create a useful reachability approximation that can power a fully automatic verification framework. Without this systematisation, the user of such frameworks would have the unpleasant task of hand picking some combination of subsystems. Note there are exponentially many combinations to choose from.

To demonstrate how local analysis can give rise to useful verification frameworks, we use 2-reachability to create a framework, called *Pair*, that checks deadlock freedom for distributed and concurrent systems. Broadly speaking, Pair is the result of replacing exact reachability for 2-reachability. This replacement makes it an approximate

framework: Pair either shows deadlock freedom or it produces an inconclusive result in the sense that it finds a state that is blocked and 2-reachable but might not be reachable. This framework improves, in terms of precision, on current local-analysis-based approximative techniques for deadlock freedom. For instance, some of the so-called non-hereditary deadlock-free systems (i.e. deadlock-free systems that have a deadlocking subsystem), which are neglected by most incomplete techniques, can be proved deadlock free by our framework. This improvement, however, comes with a price. While traditional approximate frameworks are based around polynomially checkable conditions, the problem Pair solves is NP-complete. So, we rely on SAT checkers to efficiently implement Pair in a way that, typically, scales better than current techniques for deadlock analysis.

Local analysis brings scalability at the cost of imprecision. k-reachability gives rise to frameworks that should be more scalable than exact frameworks but they cannot show properties that depend on some invariant of subsystems of size greater than k. For instance, Pair cannot show deadlock freedom if it depends on some local invariant emerging from how triples or larger combinations of components behave. So, to increase Pair's precision, we propose the *PairPicking* framework. It improves on 2-reachability by combining Pair with subsystem reachability for some subsystems picked by the user. This extension should be useful when proving deadlock freedom involves invariants of triples or larger combinations of components, these combinations can be easily identified by the user, and they are much smaller than the entire system.

This chapter's outline is as follows. Section 3.2 introduces the notion of subsystem reachability to capture and implement local analysis. In Section 3.3, we propose Pair, a strategy that uses local analysis for proving deadlock freedom. Section 3.4 introduces PairPicking, a strategy that extends Pair fully-automatic use of local analysis with some user inputs. Finally, in Section 3.5, we present our concluding remarks.

3.2 Subsystem reachability

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

The notion of *subsystem reachability* is intended to be a means to capture and implement local analysis. We analyse the behaviour of a subsystem based on the following *subsystem projection*.

Definition 3.1. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, e_{ss} the event tuple resulting from removing e_i if $i \notin ss$, and $\langle L_1, \ldots, L_n \rangle_{ss}$ the sequence of components resulting from removing the elements for which indices are not in ss. The *subsystem projection* of machine $\mathcal{S}$ over subsystem $ss \subseteq \{1 \ldots n\} \mid ss \neq \emptyset$ is given by the following supercombinator machine:

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

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

We use this projection to define *subsystem reachability*.

Definition 3.2. Let $\mathcal{S}_{ss}$ be a supercombinator machine, resulting from the projection of $\mathcal{S}$ on subsystem ss, and $(S_{ss}, \Sigma_{ss}, \Delta_{ss}, \hat{s}_{ss})$ its induced LTS. We define $reachable_{ss}(s)$, for $s \in S_{ss}$, as the reachability predicate considering this projection/subsystem's induced LTS.

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

Definition 3.3. Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and ss a given subsystem of $\mathcal{S}$. For $s \in S$, we use $reach_{ss}(s) = reachable_{ss}(s_{ss})$ to lift $reachable_{ss}$ to the states of $\mathcal{S}$. For $s = (s_1, \ldots, s_n)$, s_{ss} creates a state-tuple with the $|ss|$ elements s_i where $i \in ss$.

The over-approximating nature of this predicate is a consequence of our projection's rules discarding the participation of components outside the subsystem. This discarding can be seen as placing the subsystem in an ideal synchronisation context, where the original components outside the subsystem are replaced by components that always offer all events. So, if a projection cannot reach a state with all this help, no

combination of components (and, in particular, the original components outside the subsystem) can help the subsystem in reaching this state. By contraposition, it must be the case that if a state is reached by the system, the projected state must be reachable in the context of the projection.

Lemma 3.4. $\mathit{reachable}(s) \Rightarrow \mathit{reach}_{ss}(s)$

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

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

□

3.2.1 k-reachability

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

Definition 3.5. Let $\mathcal{S}$ be a supercombinator machine with n components and rules $\mathcal{R}$. $\mathcal{S}$'s communication graph CG is an undirected graph where nodes are component indices and there is an edge between two component indices if they participate together on a rule: $CG = (\{1 \dots n\}, \{(i,j) \mid i,j \in \{1 \dots n\} \wedge ((e_1, \dots, e_n), a) \in \mathcal{R} \wedge e_i \neq - \wedge e_j \neq -\})$.

We use connections to identify which components interact (and, consequently, directly interfere) with each other. So, we pick sets of k closely-interacting components, namely, we choose all k-sized subsystems for which component indices induce a *connected* communication subgraph. Given their close relationship, we believe these subsystems best approximate the reachability of the entire system. Note that if some (graph-theoretic) connected component of the communication graph involves only nc system components where $nc < k$, the system components in this subgraph can never be part of a set of k closely-interacting system components. Hence, in such cases, we add the nc-sized set of system components in this subgraph to our analysis. We collect subsystems in the set SS_k.

Definition 3.6. Let $\mathcal{S}$ be a supercombinator machine, CG its communication graph, $\{CG_1, \dots, CG_m\}$ this graph's connected components where $CG_i = (V_i, E_i)$, and $k_i = min(|V_i|, k)$. SS_k selects closely-interacting subsystems of size (up-to-)k:

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

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

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

Definition 3.7. Let $\mathcal{S}$ be a supercombinator machine, and SS_k its set of subsystems involving closely-interaction components.

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

The fact that this predicate soundly approximates reachability for the systems under analysis follows from Lemma 3.4 and the fact that a conjunction of over-approximations is also an over-approximation.

Lemma 3.8. $reachable(s) \Rightarrow reach_k(s)$

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

Lemma 3.9. *Let $\mathcal{S}$ be a supercombinator machine with n components and rules $\mathcal{R}$, and L_{Max} be the component with the largest LTS. For a given state s of this system and an integer k, we can test $reach_k(s)$ in time $\mathcal{O}(n^k \cdot |L_{Max}|^{k+1} \cdot |\mathcal{R}|)$.*

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

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

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

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

Lemma 3.10. $reach_{k+1}(s) \Rightarrow reach_k(s)$

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

ensures that if $ss \subseteq ss'$, $\neg reach_{ss}(s)$ implies $\neg reach_{ss'}(s)$. So, thanks to (i) and (ii), $\neg reach_{ss'}(s)$ holds, and (iii) implies that $\neg reach_{k+1}(s)$, a contradiction. □

For n the size of the system being analysed, we have that $reach_n$ is the exact reachability predicate ($reachable$). Note that either all components are connected or the system is composed of smaller (mutually disjoint/disconnected) subsystems. In the former case, the entire system is the subsystem analysed to estimate reachability, and so $reach_n$ is trivially $reachable$. In the latter case, the smaller (disjoint) subsystems forming the system are analysed separately. Since they do not affect/interfere with each other, their subsystem reachability can be independently combined to exactly capture system reachability. That is, for these subsystems reachability can be analysed as if they were completely independent systems.

Any approximation in this family is intrinsically incomplete/imprecise thanks to the pure use of local analysis. For any given *fixed* k, our notion of k-reachability is unable to show that a system respects a given property if it emerges from the behaviour of a subsystem of size greater than k. This limitation is inherent to any verification framework based on pure local analysis.

3.3 Pair: 2-reachability for deadlock freedom

In this section, we demonstrate how local analysis can be used to create an effective verification framework. We use 2-reachability to create an approximate framework to check deadlock freedom for distributed systems. We call this framework *Pair*, as it is based on the analysis of pairs of components to approximate reachability and compute blocked states. Instead of looking for cycles of dependencies between components of a system as traditional approximate frameworks do, Pair looks for states of the system that are 2-reachable and in which all further actions are blocked; a system state of this sort is considered a potential deadlock (or, deadlock candidate) and we call it a *Pair candidate*.

Definition 3.11. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. A state $s = (s_1, \ldots, s_n) \in S$ is a *Pair candidate* if and only if $pair_candidate(s)$ holds, where $pair_candidate(s) \mathrel{\widehat{=}} reach_2(s) \wedge blocked(s)$.

Our framework is sound, as absence of Pair candidates implies deadlock freedom. This follows from the fact that $reach_2(s)$ approximates reachability as per Lemma 3.8.

Theorem 3.12. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. If $\mathcal{S}$ is pair_candidate free, it must be also deadlock free.*

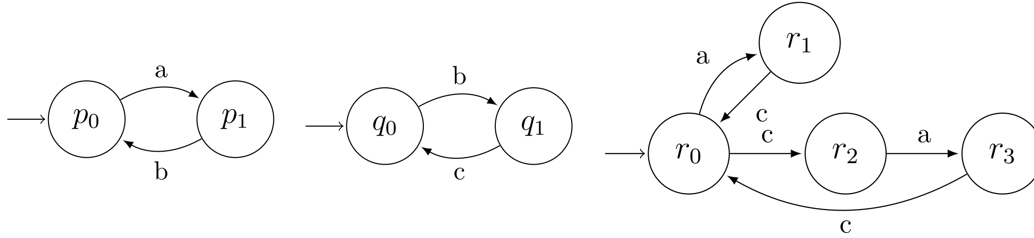

Figure 3.1: LTSs of components L_1, L_2 and L_3, respectively.

This criterion will be shown to be more accurate than many current approximate frameworks that check deadlock freedom, but it remains incomplete as it relies on local analysis to approximate reachability; there may well be Pair candidates that are not actually reachable.

Example 3.1. Let $\mathcal{S} = (\langle L_1, L_2, L_3 \rangle, \mathcal{R})$ be the supercombinator machine such that the components are described graphically in Figure 3.1 and they must synchronise on shared events. That is, $\mathcal{R} = \{((a, -, a), a), ((b, b, -), b), ((-, c, c), c)\}$.

For this system, the state (p_0, q_0, r_3) is 2-reachable and blocked, but not reachable. Thus, it constitutes a Pair candidate but not a deadlock. ■

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

3.3.1 Precision of Pair

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

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

Definition 3.13. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. The *pairwise machine* $\mathcal{S}_{i,j}$ is used to analyse the interactions of components i and j.

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

In Martin's approach, a dependency digraph is constructed and then analysed for absence of cycles. The dependency digraph constructed has a node for each state of each component, and an edge from a state s of component i to a state s' of component j if and only if $reachable_{i,j}((s, s'))$ and $ungranted_request_{i,j}(s, s')$ hold, where $reachable_{i,j}$ denotes the *reachable* predicate for the LTS induced by $\mathcal{S}_{i,j}$, and $ungranted_request_{i,j}(s, s')$ holds when, in their respective states (i in s and j in s'), component i is willing to interact (i.e. engage on a rule) with j (according to $\mathcal{S}_{i,j}$) but j is unable to do so.

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

Definition 3.14. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$. Let $\mathcal{U}$ be the disjoint union of all S_i and $s_{i,j}$ denotes state j of the component i. A SDD candidate is a sequence of component states $c \in \mathcal{U}^*$ where for all $i \in \{0 \ldots |c| - 1\}$, given that $c_i = s_{j,k}$ and $c_{i \oplus 1} = s_{l,m}$, $reachable_{j,l}((s_{j,k}, s_{l,m}))$ and $ungranted_request_{j,l}(s_{j,k}, s_{l,m})$ hold, where $\oplus$ is addition modulo $|c|$.

This method can carry out deadlock-freedom verification very efficiently: a digraph can be shown to have no cycles in linear time using a modified depth-first search. This efficiency, however, comes with a price as the use of a cycle as a candidate makes this method imprecise in several ways. Firstly, a cycle might not be consistent with basic sanity conditions such as it must have a single node per component (after all no component can be in two different states in a single deadlock). Secondly, a cycle is only partially consistent with the local reachability and local blocking properties derived from the analysis of pairs of components. Note that only adjacent elements in the cycle are guaranteed to be pairwise reachable and pairwise blocked. So, there may be local properties of non-adjacent component states not tested for that might eliminate some SDD candidate. For instance, a cycle where two non-adjacent component states are not mutually reachable (or, can effectively synchronise) cannot represent a true

deadlock and so it should not be considered a SDD candidate. Finally, a cycle, as a necessary condition, is bound to arise in some deadlock-free systems. Thus, in such cases, this framework is ineffective. The reason why these sources of imprecision are not addressed is that these methods look for polynomially checkable conditions for guaranteeing deadlock freedom and tackling any of these sources of imprecision is likely to make the problem of finding a candidate *NP*-hard.

The combination of 2-reachability and exactly checking for blocked system states makes Pair more precise than SDD both in terms of reachability and the blocking conditions. Pair uses an *exact* characterisation for blocked states instead of the imprecise cycle-of-dependencies one. As for reachability, SDD only requires pairs of component states adjacent in the cycle to be mutually reachable. Pair, however, enforces this for all pairs of components in a blocked system state. The improvement in precision means that none of the potential sources of imprecision highlighted in the previous paragraph affects Pair. In fact, it is only imprecise in terms of reachability. The use of 2-reachability means that it cannot prove deadlock freedom if this property depends on some reachability invariant of triples or larger combinations of components.

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

Lemma 3.15 (Theorem 1. in [Mar96])**.** *Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. Let $s = (s_1, \ldots, s_n) \in S$ be a system state. If blocked(s) holds, there must be a cycle $c \in \{s_1, \ldots, s_n\}^*$ such that there is an ungranted request from each c_i to $c_{i\oplus 1}$, where $\oplus$ is addition modulo $|c|$.*

It follows from this lemma and the fact that Pair candidates are 2-reachable that we can construct a SDD candidate from any Pair candidate.

Lemma 3.16. *Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. Let $s = (s_1, \ldots, s_n) \in S$ be a system state. If pair_candidate(s) holds, there must be a cycle $c \in \{s_1, \ldots, s_n\}^*$ such that there is an ungranted request from each c_i to $c_{i\oplus 1}$ and c_i and $c_{i\oplus 1}$ are pairwise reachable, where $\oplus$ is addition modulo $|c|$.*

This lemma, in turn, implies by contraposition that a SDD-candidate-free system must also be Pair-candidate-free.

Corollary 3.17. *Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $\mathcal{U}$ the disjoint union of all the component states of each component.*

$$\neg\, \exists\, c : \mathcal{U}^* \bullet sdd_candidate(c) \Rightarrow \neg\, \exists\, s : S \bullet pair_candidate(s)$$

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

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

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

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

Each component in such a system can alternate in behaving as a client or a server performing request and response actions. As a client the component must be able to

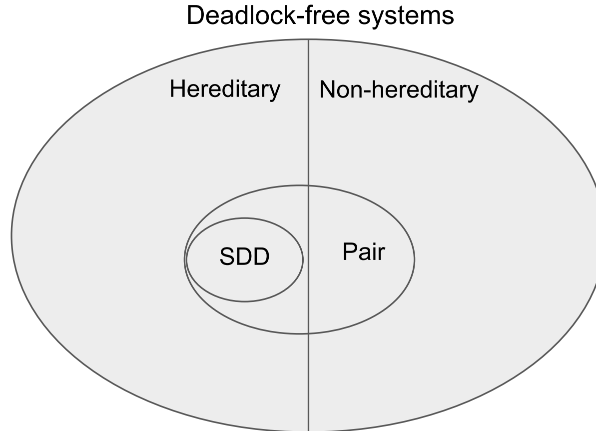

Figure 3.2: The relationship between deadlock-freedom-checking frameworks.

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

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

This analysis points to a class of relevant systems that can be proved deadlock free by Pair but not by SDD: the class of *non-hereditary deadlock-free systems*. These systems have a subsystem that can deadlock and a guard-like component that leads the subsystem away from this blocked state. SDD cannot show such systems deadlock free since if a subsystem deadlocks then there must exist a cycle of ungranted requests between the states of components in this subsystem that constitutes a SDD candidate as per Lemma 3.15. While SDD only allows cycles of ungranted requests to be broken by component states within the cycle, Pair can be understood as allowing

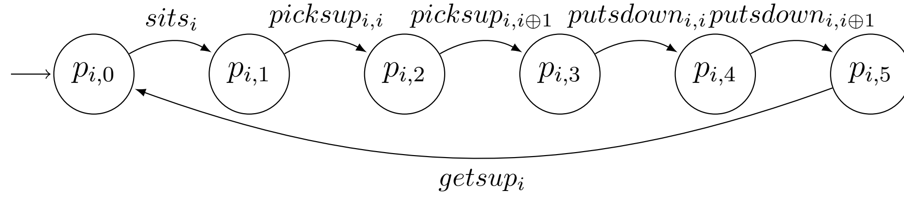

Figure 3.3: LTS of philosopher i.

also component states outside the cycle to break it. Hence, its ability to tackle such systems. Roughly speaking, SDD can be seen as a method that tries to prove *hereditary* deadlock freedom (i.e. that no subsystem can deadlock) using local analysis. On the other hand, our method can prove deadlock freedom for both hereditary and non-hereditary deadlock-free systems, such as the following example. We point out that many concurrent and distributed systems are non-hereditary deadlock-free; systems that have components implementing mutual exclusion algorithms or semaphores to prevent some subsystem from reaching undesired states are fairly commonplace in the concurrency literature. Figure 3.2 illustrates the relative completeness of SDD and Pair, that is, the relationship between the set of deadlock-free systems Pair can prove deadlock free and the set that SDD can.

Example 3.2. This well-known system is composed of three different types of components: forks, philosophers and a butler. We parametrise our system with N, which denotes the number of philosophers in the system.

A philosopher has access to a table at which it can pick up two forks to eat: one at its left-hand side and the other at its right-hand side. A fork is placed, and shared, between philosophers sitting adjacently in the table. So, in this system, N philosophers have access to a table, where they can sit and compete to acquire two forks in order to eat. The butler is in charge of managing the access of philosophers to the table. The behaviour of philosopher (fork) i is depicted in Figure 3.3 (3.4). We use $\oplus$ to denote addition modulo N.

Given that these components synchronise on their shared events, the philosophers and forks can reach a deadlock state in which all philosophers have acquired their left-hand side forks and, as a consequence, no right-hand side fork is left to be acquired. The butler is introduced to prevent all the philosophers from sitting at the table at the same time, thereby precluding this deadlock state. We use b_S to depict the state in which the butler has allowed the philosophers in S to the table. So, the butler states space is given by the set of all b_S where $S \in \mathbb{P}(\{1 \ldots N\}) - \{\{1 \ldots N\}\}$. Its transitions are created as depicted in Figure 3.4, and its initial state is given by $b_\emptyset$.

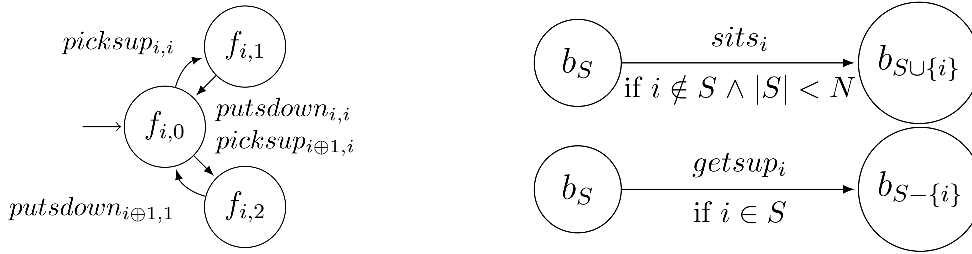

Figure 3.4: LTS of fork i and transitions of the butler process.

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

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

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

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

3.3.2 Complexity of Pair

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

Next, we show that the problem of detecting a Pair candidate is *NP*-complete, and we discuss some implications of this result.

Theorem 3.18. *Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. The problem of deciding $\exists\, s : S \bullet pair_candidate(s)$ is NP-Complete.*

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

We show (i) by establishing that, for a given system state s, $pair_candidate(s)$ can be verified in $\mathcal{O}(n^2 \cdot |L_{MAX}|^3 \cdot |\mathcal{R}|)$ time, where n and $\mathcal{R}$ gives the number of components and the rules of the system, respectively, and $|L_{Max}|$ the size of the largest component LTS. $reach_2$ can be checked in $\mathcal{O}(n^2 \cdot |L_{Max}|^3 \cdot |R|)$ time, as per Lemma 3.9, while $blocked(s)$ can be checked in $\mathcal{O}(|\mathcal{R}| \cdot n \cdot |L_{Max}|)$ time. To check $blocked(s)$, one can simply check whether a rule can be applied to s.

To demonstrate (ii), we present a polynomial-time reduction from the CNF-SAT problem to our Pair-candidate detection problem. To begin with, we introduce the CNF-SAT problem and some useful notation.

Definition 3.19. Given a CNF boolean formula $\mathcal{F}$ with boolean variables $x_1, \ldots, x_m$, the *CNF-SAT* problem consists of finding an assignment to the m boolean variables so that $\mathcal{F}$ holds. That is, checking $\exists\, x_1, \ldots, x_m : \{true, false\} \bullet \mathcal{F}$.

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

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

In this machine, component F_i captures the satisfiability of clause $\mathcal{F}_i$, whereas component X_i models the assignment of boolean variable x_i. We use literals x_j and $\neg x_j$ as events that denote whether variable x_i has been assigned to $true$ and $false$, respectively. So, in particular, note $\mathcal{F}_{i,j}$ are events. Satisfiability of $\mathcal{F}_i$ is captured by creating a transition, in F_i, with event x_j ($\neg x_j$) to a *terminal* deadlocked state for each literal x_j ($\neg x_j$) in $\mathcal{F}_i$. So, a terminal state is only reached if the clause has been satisfied (i.e. a literal has been satisfied). Next, we present the definitions of component F_i and X_i and their graphical representation in Figure 3.5.

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

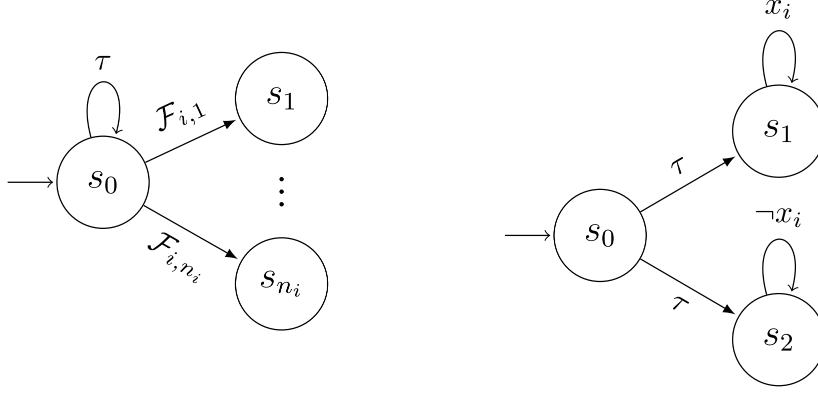

Figure 3.5: LTSs F_i and X_i respectively.

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

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

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

The set of rules provided enables the synchronisation between variable components and clauses components so that the satisfiability of the clauses is guided by the assignment of variables. For convenience, we use our squared-bracketed notation to represent an event tuple as a set of pairs. Also, we use $F_i = i$ and $X_{i,j} = n + m_{i,j}$ to conveniently denote the positions of components in the event tuple, where $m_{i,j}$ denotes the index of the boolean variable in literal $\mathcal{F}_{i,j}$, so if $\mathcal{F}_{i,j} = x_k$ or $\mathcal{F}_{i,j} = \neg x_k$ then $m_{i,j} = k$.

$$\begin{aligned}\mathcal{R} = \; & \{([i, \tau], \tau) \mid i \in \{1 \ldots n + m\}\} \\ \cup & \bigcup_{i \in \{1 \ldots n\}} \{([(F_i, \mathcal{F}_{i,j}), (X_{i,j}, \mathcal{F}_{i,j})], \mathcal{F}_{i,j}) \mid j \in \{1 \ldots n_i\}\}\end{aligned}$$

From $\mathcal{S}$'s definition, we can see this supercombinator machine can be constructed in polynomial time on the size of the formula $\mathcal{F}$. All components can be constructed in time proportional to $|\mathcal{F}|$. Each component X_i can be constructed in time $\mathcal{O}(1)$, whereas

components F_i can be constructed in time $\mathcal{O}(|\mathcal{F}_i|)$. Rules $\mathcal{R}$ can be constructed in time $\mathcal{O}(|\mathcal{F}|)$, as it creates a rule per literal and a τ-rule per component.

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

If the formula is satisfiable, $\mathcal{S}$ has a Pair-candidate. Let $\mathcal{A}$ be a satisfying assignment for the formula. The system $\mathcal{S}$ can reach a system state where components X_i are in states respecting their valuation $\mathcal{A}(x_i)$, and components F_i are in terminal states. This state is reachable as components X_i can simply τ-transition to their respective valuation states, and because $\mathcal{A}$ is a satisfying assignment, each component F_i must be able to synchronise with some X_i to reach a terminal state. If a state is reachable it is also 2-reachable. This state is also blocked since all F_i components are in terminal states and all components X_i are in states that can only trigger system rules involving the participation of at least one F_i component.

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

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

Corollary 3.20. *Let $reach(s)$ be a reachability over-approximation. If $reach(s) \Rightarrow reach_2(s)$ then the problem of detecting a deadlock candidate s such that $reach(s) \wedge blocked(s)$ is* NP*-hard.*

This corollary, Lemma 3.9 and the fact that $blocked(s)$ can be decided in polynomial time, as showed in this proof, imply that any deadlock candidate formulation using k-reachability, for $k \geq 2$, instead of exact reachability gives rise to an *NP*-complete candidate detection problem.

Corollary 3.21. *For integer $k \geq 2$, the problem of detecting a deadlock candidate s such that $reach_k(s) \wedge blocked(s)$ is* NP*-complete.*

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

3.3.3 Pair-candidate detection via SAT solving

As demonstrated in Section 3.3.2, the problem of detecting a Pair candidate is *NP*-complete. So, currently, we only know of deterministic procedures that take exponential time to solve it. There have been, however, some remarkable advances in proposing efficient procedures to solve the propositional satisfiability (SAT) problem. So, in an attempt to efficiently tackle Pair-candidate detection, we propose an implementation for our framework where we translates Pair-candidate detection into propositional satisfiability, which can later be checked by a SAT solver.

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

The formula *Candidate* is composed of three sub-formulas: *State*, $Reach_2$, and *Blocked*.

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

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

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

Sub-formula $Reach_2$ captures the approximation $reach_2$. To describe this sub-formula (and the next one), we introduce the following notation for convenience. S_{ss} gives the state space of subsystem ss. We represent subsystem states $sst \in S_{ss}$ by a set of pairs, as opposed to the traditional tuple representation, where $(i, s) \in sst$ denotes that state s of component i belongs to this subsystem state. To capture $Reach_2$, we examine the state space of subsystems in SS_2 and disallow unreachable combinations of component states. For instance, if by analysing how components i and j interact, we discover that they cannot reach their respective states s and s' simultaneously, we add the constraint $\neg st_{i,s} \vee \neg st_{j,s'}$.

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

The sub-formula *Blocked* captures the *blocked* predicate. We assume triple-disjointness of the input system in encoding this sub-formula; this is the only definition in this entire chapter that formally relies on this assumption. Thanks to triple disjointness, we can capture whether a system state is blocked by analysing only individual and pairs of components – after all, system transitions only involve either the participation of a single component or a pair of them. To encode this blocking requirement, we rely on $\mathcal{S}'_{ss} = (\langle L_1, \dots, L_n \rangle, \mathcal{R}_{ss})$, a slightly modification of our projection supercombinator machine, where $\mathcal{R}_{ss} = \{(e, a) \mid (e, a) \in \mathcal{R} \wedge \forall i : \{1 \dots n\} \bullet (i \notin ss \wedge e_i = -) \vee (i \in ss \wedge e_i \neq -)\}$. Unlike the original projection, this one does not truncate rules so it only uses rules that require the exact participation of components in ss. We use $s \longrightarrow_{ss}$ to denote that there exists a rule in $\mathcal{S}'_{ss}$ that can be triggered from state s. In this encoding, we examine the state space of subsystems $ss \in SS_1 \cup SS_2$ according to projection $\mathcal{S}'_{ss}$ and forbid component states in subsystem states that can engage in some rule from holding simultaneously.

$$Blocked \mathrel{\widehat{=}} \bigwedge_{ss \in SS_1 \cup SS_2} \bigwedge_{\substack{sst \in S_{ss} \wedge \\ sst \longrightarrow_{ss}}} (\bigvee_{(i,s) \in sst} \neg st_{i,s})$$

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

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

3.3.4 Practical evaluation

We implement our framework in the *DeadlOx* tool: given a supercombinator machine, it produces our SAT encoding which is then checked by the Glucose 4.0 solver [AS09]. The implementations of all frameworks we propose follow the same architecture. We use FDR4, as a library, to translate/compile CSP models into supercombinator machines. Given this supercombinator machine, our tools proceed to construct the SAT/SMT encoding proposed for the framework at hand. This encoding is, then, solved by a SAT/SMT solver to check whether an appropriate candidate exists; our tools rely on the Glucose 4.0 SAT solver [AS09] and the Z3 SMT solver [dMB08]. Some of the frameworks presented in later chapters are based on a SMT encoding of their candidate detection problem. We do use FDR4 to translate CSP models into supercombinator machines. However, any other language could be used if a translation into supercombinator machines is provided. Also, our frameworks could be implemented on top of any other SAT/SMT solvers. The code base for implementing all tools in this thesis consists of about 10k lines of C++ code. DeadlOx and the models used in this section are available in our experiment package [AGRR18]. In fact, this package contains binaries for all tools that we implement in this thesis and all the input examples we use to test our frameworks.

In this section, we evaluate our framework and DeadlOx tool. We extend the input language of FDR4 with the annotation `:[Pair]` that should be added to a deadlock free assertion. It tells FDR4 to use our Pair technique instead of explicit state exploration to check the assertion. For instance, a distributed system described by process `SYSTEM` could be checked using Pair by the following assertion.

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

Our experiment evaluates deadlock freedom for some triple-disjoint deadlock-free systems. The experiment was conducted on a dedicated machine with a quad-core

Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. We compare DeadlOx against the Deadlock Checker [MJ97], FDR4's deadlock freedom assertion (FDR) [GRABR14], and D-Finder 2 [BGL+11]. Deadlock Checker implements the SDD framework, FDR4 is a complete/exact method that performs explicit state exploration, and D-Finder 2 is an approximative approach that uses some tailored system invariants. D-Finder 2 implements three techniques to calculate these invariants: a boolean-constraint-based (DF2pm), a fixed-point-based (DF2fp), and a enumerative one (DF2l). Also, when appropriate, we combine FDR4's explicit state exploration with partial order reduction (FDRp) [GRHRW15] or compression techniques (FDRc) [RGG+95].

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

Table 3.1 presents the results for the hereditary deadlock-free systems. As for approximative methods, these results suggest that our method scales similarly to SDD. Pair can show deadlock freedom for both Tel and Tarry examples while SDD cannot. This demonstrates what we informally claimed when we compared Pair and SDD, namely, Pair is better than SDD even for hereditary deadlock-free systems. D-Finder 2's approaches, although approximate, seem to be much less efficient than any other method, even complete ones. It seems that the calculations of invariants 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.

As for exact methods, Pair fares better than FDR4's techniques for most examples. For Lock, Tarry and SWP, FDR4's explicit exploration, however, fares better. For these examples, the system's state space does not grow so rapidly as the number of components increases, hence, FDR4's good performance. For Tarry and SWP, the state space of individual components grows rather drastically with the increase of N. Pair's quadratic blow up in analysing pairs of components combined with this rapid growth is the reason behind Pair's lack of scalability. We point out that the combination

of FDR4's deadlock-free assertion with compression techniques can be remarkably efficient for some systems. The use of compression, however, requires manually devising a compression strategy, whereas all other methods are fully automatic. Also, a lot of skill is necessary in devising effective strategies.

Table 3.2 presents the results for the non-hereditary deadlock-free systems. SDD (and any framework based on the detection of cycles of ungranted requests) is unable to show deadlock freedom for systems in this class. Pair, on the other hand, can. It can show deadlock freedom for these three variants of the butler solution to the dining philosophers problem. ButI is a solution with a single butler that allows only $N-1$ philosophers to sit at the table, where N is the total number of philosophers, and it does that by keeping track of the identity of philosophers sat at the table. Hence, the state space of this butler component grows exponentially as N increases. This growth is the reason why Pair does not scale for this example. ButID circumvents this problem by having $N-1$ butlers each of which allow a philosopher to the table. Pair does not scale well for this example either. As N grows, the size of individual components increases linearly, the number of components increase linearly, and the number of edges in the communication graph grows quadratically. So, Pair's lack of scalability comes from the fact that it needs to analyse a quadratic number of pairs of components and each of these analyses creates a constraint that is quadratic on the size of components. Finally, ButID2 creates a solution that is more amenable to Pair. In this solution, we have $N-1$ butlers but each of them only takes care of 5 fixed philosophers. In this setting, as N grows, the size of components remains constant, the number of components increases linearly, and the number of connections in the communication graph grows linearly. These solutions are different from the more traditional one where a butler simply counts the number of philosophers sat at the table regardless of their identity. This traditional solution, however, cannot be tackled by Pair as it requires global analysis of the system. Intuitively, by adding the identity of philosophers we transform the global invariant "a philosopher must be left out of the table at all times" into a pairwise one that Pair can capture.

Unsurprisingly, Pair is not able to prove deadlock freedom when this property depends on some global invariant preserved by the system (or perhaps by larger subsets of the system than the pairs used here). For instance, proving deadlock freedom for Milner's scheduler, which is a fairly simple well-known system, is out of our method's reach. The issue with Milner's scheduler is that it is essentially a token ring for which deadlock freedom depends on the fact that there is always precisely one token present; this latter property cannot be proved by local analysis of the sort we employ. We

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

Table 3.1: Results for hereditary deadlock-free 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 some error occurred, such as running out of memory. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found.

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

Table 3.2: Results for non-hereditary deadlock-free 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 some error occurred, such as running out of memory. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found.

point out, however, that techniques presented in later chapters can be used to verify this system.

3.4 PairPicking: Pair meets user's picks

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

Pair is a fully automatic framework designed around the analysis of pairs of components, and as such, it cannot prove deadlock freedom when this property depends on the behaviour of triples or larger combinations of components. In PairPicking, we try to tackle this imprecision by giving the user the ability to manually identify some subsystems, involving more than two components, they believe are necessary in proving deadlock freedom. The reachability approximation derived from these subsystems are combined with $reach_2$. Unlike Pair, we do not calculate the reachability approximation for a subsystem and lift it to the entire system, instead, we replace the chosen subsystems by their projections. We rely on the following supercombinator

machine to carry out this substitution. Note that the following definition requires rules to have distinct system events. A system can be converted to this format by changing rules with the same system events, so they all have distinct ones. This change does not introduce or remove deadlock states of the original system, and it can be carried out efficiently by simply iterating over the system's rules.

Definition 3.22. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where rules in $\mathcal{R}$ have *distinct* system events (i.e. if (e, a) and (e', a) are members of $\mathcal{R}$ then $e = e'$), and subsystems $ss_1, \ldots, ss_m$ partitioning the system, i.e. where $\bigcup_{i \in \{1 \ldots m\}} ss_i = \{1 \ldots n\}$ and $ss_i \cap ss_j = \emptyset$ for $i \neq j$ hold. $\mathcal{S}_{ss_1,\ldots,ss_m}$ is the supercombinator machine where the combination of components in each subsystem ss_i is replaced by its projection's LTS.

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

- L_{ss} the LTS induced by $\mathcal{S}_{ss}$ as per Definition 3.1.
- $\mathcal{R}' = \{([(i, a) \mid i \in \{1 \ldots m\} \wedge \exists j : ss_i \bullet e_j \neq -], a) \mid ((e_1, \ldots, e_n), a) \in \mathcal{R}\}$

Since we make sure rules are adapted to enforce the same interactions as the original does, this new supercombinator machine ends up deadlocking whenever the original does. Thus, we can freely replace a supercombinator machine by this modified version when checking for deadlock freedom. In particular, we can apply Pair to this modified machine instead; this application is exactly what constitutes the PairPicking strategy.

Theorem 3.23. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where rules have distinct system events, and $ss_1, \ldots, ss_m$ a set of subsystems such that they partition the system. $\mathcal{S}_{ss_1,\ldots,ss_m}$ deadlocks iff $\mathcal{S}$ does.*

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

This modified version of a supercombinator machine does not alter its behaviour but it does alter its structure. When applied to this machine, Pair would consider, as this machine does, the parallel combination of components in each subsystem as an individual component. So, by analysing the overall behaviour of these combinations, Pair is implicitly making use of reachability approximations induced by these subsystems. Therefore, the application of Pair to such a modified machine where triples or larger combination of components are put together creates a framework that precisely

understands how these larger subsystems work and, thus, is more precise in showing deadlock freedom.

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

The complexity of this strategy can be as bad as explicit state space analysis of the entire system; after all we do not forbid the user from choosing the entire system as a subsystem. Nevertheless, it should be clear that by picking small subsystems, this strategy is only explicitly analysing small state spaces. Hence, in practice, if small subsystems are chosen, this strategy should considerably outperform explicit state space exploration of the overall system.

3.4.1 Examples and practical evaluation

In this section, we introduce two families of systems and discuss how to apply PairPicking to them. The PairPicking strategy uses our implementation of Pair and the built-in function `explicate` of FDR4; `explicate` is used to create the modified system that is in turn checked deadlock free by Pair.

The `explicate` function can be used to construct the induced LTS for a given subsystem. Let us assume that a system is described by the distributed-alphabetised-parallel combination $\|_{i=0}^{N} (A_i, C_i)$ (see Section 2.1 for details on this operator). To compute the induced LTS for subsystem $\{j, \ldots, k\}$, one can use the `explicate` function on the distributed-alphabetised-parallel combination of its components, that is, we could use this function to create process $ss_1 = \texttt{explicate}(\|_{i \in \{j,\ldots,k\}} (A_i, C_i))$. This process ss_1 is a single internal FDR4 component that represents the induced LTS of this subsystem. Such induced LTSs can, then, be combined using the distributed-alphabetised-parallelism operator to construct the modified system. For instance, the pair $(\bigcup_{i \in \{j,\ldots,k\}} A_i, ss_1)$ would represent the subsystem $\{j, \ldots, k\}$ in the sort of modified system PairPicking uses.

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

components. Our two families of systems differ on the way local networks choose their interface component.

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

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

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

Table 3.3 presents the results for the voting-based family of systems, whereas Table 3.4 presents the results for systems in the priority-based family. The name of each example describes the topology used to connect local networks and the number of components in each of them. For instance, VGrid4 is the system where local networks are connected in a grid-like fashion and each of them is composed of 4 components. These results show that PairPicking is quicker in showing deadlock freedom for these systems than complete approaches. FDR4's assertion combined with compression techniques comes close to the sort of speed PairPicking achieves. We believe, however, that the sort of manual work needed to use PairPicking, namely, selecting the subsystems that need to be explicit examined, is much less complex than the work needed to craft a compression strategy.

These results also confirm the practical limitations of PairPicking. Since it performs explicit exploration to create the LTSs of the chosen subsystems, it suffers with the

state-space explosion problem. So, even for small subsystems, the size of their LTSs tends to make the verification cost prohibitive – even for approximative approaches such as Pair. Furthermore, these results also demonstrate how the topology affects the underlying use of Pair. A system with a fully-connected topology normally suffers with two sources of complexity as the number of components grows. Firstly, with the addition of a component, there is an increase in the complexity of individual components as they have to account for the communication with this newly included component. Secondly, the number of connections between components, and of the pairs of components to explicitly analyse, grows quadratically with the linear increase of components. On the other hand, for communication topologies where each component is connected to a fixed number of components, such as grids, rings or chains, neither of these two problems arise. These factors help explain the difference in scalability for the different systems (and topologies) we have analysed.

3.5 Conclusion

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

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

We use this 2-reachability to create Pair, an approximate framework that checks deadlock freedom. The use of this approximation tackles some sources of imprecision of traditional techniques that check for cycles of dependencies. So, it represents an improvement in the accuracy of current approximate techniques; in particular, some non-hereditary deadlock-free systems, which are neglected by most approximate techniques, can be tackled by our framework. This improvement comes at a price. The problem tackled by Pair is *co-NP*-complete, whereas traditional approaches rely on conditions that can be checked in polynomial time. Still, Pair-candidate detection

		Approximate	Exact		
Example	N	PP	FDR	FDRc	FDRp
VChain4	10	0.47	*	1.52	*
	20	0.46	*	5.98	*
	30	0.66	*	19.95	*
	40	0.87	*	56.63	*
VGrid4	10	0.52	*	*	*
	20	1.27	*	*	*
	30	2.22	*	*	*
	40	2.62	*	*	*
VFully4	5	0.61	*	*	24.30
	8	2.02	*	*	*
	12	8.73	*	*	*
	15	21.56	*	*	*
VChain5	10	1.22	*	27.92	*
	20	2.07	*	69.40	*
	30	3.12	*	135.98	*
	40	4.27	*	266.88	*
VGrid5	10	1.82	*	*	*
	20	5.58	*	*	*
	30	8.98	*	*	*
	40	12.39	*	*	*
VFully5	5	2.02	*	*	118.33
	8	8.53	*	*	*
	12	39.29	*	*	*
	15	97.15	*	*	*
VChain6	10	15.19	*	*	*
	20	31.78	*	*	*
	30	48.86	*	*	*
	40	65.74	*	*	*
VGrid6	10	25.61	*	*	*
	20	67.69	*	*	*
	30	110.18	*	*	*
	40	150.36	*	*	*
VFully6	5	20.70	*	*	*
	8	85.03	*	*	*
	12	282.70	*	*	*
	15	*	*	*	*

Table 3.3: Results for voting-based systems. N gives the number of local networks in 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.

should be easier to handle if compared to the *PSPACE*-completeness of exact deadlock checking. Intuitively, Pair needs to analyse pairs of components to approximate

		Approximate	Exact		
Example	N	PP	FDR	FDRc	FDRp
PChain4	5	0.87	*	0.92	5.18
	10	1.12	*	2.07	*
	15	1.72	*	3.27	*
	20	2.37	*	6.28	*
PGrid4	5	0.51	*	0.87	5.12
	10	2.02	*	*	*
	15	4.07	*	*	*
	20	5.83	*	*	*
PFully4	3	0.67	18.95	1.87	0.22
	5	1.62	*	*	44.44
	8	6.88	*	*	*
	12	25.47	*	*	*
PChain5	5	4.87	*	7.73	35.47
	10	10.74	*	18.05	*
	15	16.80	*	29.62	*
	20	22.91	*	44.25	*
PGrid5	5	4.72	*	7.68	35.52
	10	18.25	*	*	*
	15	32.98	*	*	*
	20	48.86	*	*	*
PFully5	3	3.72	*	41.19	0.71
	5	14.54	*	*	254.25
	8	53.26	*	*	*
	12	165.38	*	*	*
PChain6	5	47.55	*	278.26	234.85
	10	107.92	*	*	*
	15	169.48	*	*	*
	20	229.17	*	*	*
PGrid6	5	47.30	*	259.57	235.38
	10	173.46	*	*	*
	15	*	*	*	*
	20	*	*	*	*
PFully6	3	35.99	*	*	3.90
	5	128.18	*	*	*
	8	*	*	*	*
	12	*	*	*	*

Table 3.4: Results for priority-based systems. N gives the number of local networks in 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.

reachability whereas exact frameworks need to go over the system's entire state space. Despite the inherent complexity of detecting Pair candidates, SAT checkers

can efficiently implement our framework. In terms of efficiency, our implementation, typically, fares similarly to traditional approximate techniques and much better than exact frameworks. This is demonstrated by a series of practical experiments.

We have not investigated the use of k-reachability where $k > 2$. Most of the examples we have worked with were either proved deadlock free by local invariants calculated by pairwise analysis or by global invariants. Also, we could not immediately think of a relevant class of systems that would require this sort of k-reachability. Obviously, that does not mean such a class does not exist.

Pair cannot show deadlock freedom when it depends on some reachability invariant of triples or larger combinations of components. To cope with that, we propose PairPicking, a strategy that combines the reachability analysis of some hand picked subsystems with Pair. The choosing of subsystems enables the user to capture invariants of triples or larger combinations of components. This strategy should be applied when these subsystems are small and easy to identify.

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

Chapter 4

Global analysis using synchronisation analysis

4.1 Introduction

Local analysis is unable to show deadlock freedom (or any property) if it depends on some global system invariant. To address this problem, some sort of *global analysis* is needed. Explicit exploration of the system's behaviour would create a global-analysis approach too vulnerable to the state-space explosion problem. So, in this chapter, we propose techniques that implement approximate global analysis by combining invariants of individual components.

In this chapter, 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 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. This notion of cooperation consistency/feasibility for this combination captures *global invariants* of the system.

Our CSA frameworks use abstract interpretation concepts [CC77, CC79] that are normally part of data-flow-analysis frameworks [NNH99] 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 we are aware, our idea of synchronisation analysis and the techniques we implement are new.

We combine our synchronisation-analysis techniques with the Pair framework presented in Chapter 3 to create *PairStatic*. Pair uses pure local analysis to verify systems deadlock free. So, it cannot prove deadlock freedom if it depends on some global invariants of the system. PairStatic, however, should leverage the global invariants our synchronisation-analysis techniques capture to improve on Pair's deadlock-freedom checking capabilities. For instance, PairStatic can capture some global invariants that ensure deadlock freedom for some 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 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.

This chapter's outline is as follows. In Section 4.2, 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 4.3 introduces PairStatic, a framework that combines Pair's local analysis with the global-analysis techniques proposed in this chapter. Finally, in Section 4.4, we present our concluding remarks.

4.2 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 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 some types of invariants and concurrency mechanisms these consistency notions can capture.

4.2.1 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 [CC77, CC79] in a way similar to what data-flow-analysis (DFA) frameworks do; we were particularly inspired by [NNH99]. 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 4.1 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 a *family of abstract transformers* 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

Figure 4.1: Example of components L_1 and L_2.

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 the performing of any event 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; proposition (i) is the component-state invariant computed by CSA. More informally, X_{s_i} captures what (the information) we "know" about state s_i.

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 [NNH99]. 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 [Tar55] and to the finite height of the domain[1]. 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 $X_{s_0} = \{0\}$, $X_{s_1} = \{1\}$, $X_{s_2} = \mathbb{Z}$.

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 4.1. *The worklist algorithm can calculate a fixed-point X in $\mathcal{O}(|S| \cdot |\Delta| \cdot h^2)$.*

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 one equation per transition so we have $|\Delta| + 1$-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.

[1]The worklist algorithm and Knaster-Tarksi theorem can be understood as operating on the product domain D^n where n gives the number of X_{s_i} variables.

For instance, state (s_1, s_1) cannot be reached because while L_1 does not perform a to get to s_1, L_2 needs to perform a once to get to s_1. As they need to synchronise on shared events, they can only reach combinations of component states where they agree on the number of performed as. This sort of (in)consistency checking is what is behind the reachability testing implemented by synchronisation analysis we introduce next. We also point out to the imprecise nature of this sort of framework. Any system involving component state s_2 of L_2 is unreachable, as at least two as need to be performed by L_2 to reach this component state and L_1 can only perform a once whichever behaviour (path) this component engages on. Our CSA framework, however, approximates that the number of as that L_2 can perform to reach s_2 is in the set $\mathbb{Z}$; this over-approximations is a consequence of s_2's self-loop performing a. So, our example framework is, in particular, unable to prove unreachability for system states involving state s_2 of L_2.

4.2.2 Synchronisation analysis

Synchronisation analysis combines 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 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 the system state 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 through 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 involve rule r_k. The use of r_k both as a rule and an event should cause no confusion.

Definition 4.2. 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 chapter. 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.

4.2.2.1 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 4.1 as a running example to explain this technique.

Example 4.1 (From [Ros10])**.** This example describes a ring-like message-exchange system. Components route messages unidirectionally around the network. A component receives messages from its predecessor component in the ring or from its user, and it either passes the message over to the next component in the ring or outputs the message to its user. Each component can hold up to two messages at a time and the second message must have been received from its predecessor and not from its user.

This system is captured by machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ where L_0, L_1 and L_2 are defined in Figure 4.2 and $\mathcal{R}$ is 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. Component i receives a message from its predecessor component in the ring via event $ring_i$, and from its user via event in_i. It can pass a message along to the next component in the ring via event $ring_{i\oplus 1}$, and output the message to its user via out_i. The τ transitions represent an internal (non-deterministic) decision of the component.

The *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

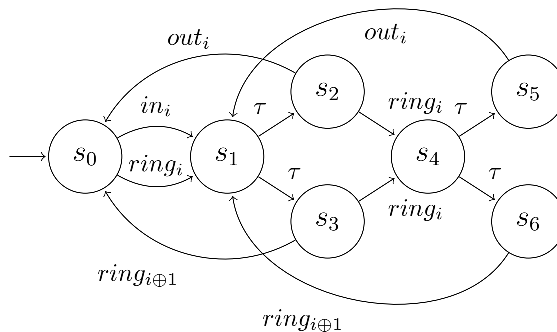

Figure 4.2: LTS of component L_i where $\oplus$ represents addition modulo 3.

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, 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'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 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 4.3. 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, $\langle\rangle$ is the top element and $\bot$ is the bottom 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 $T_{r_k}(d) = d\hat{}\langle r_k \rangle$.

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^l_k to denote the l-th most recent system-wide occurrence of rule k, namely, o^l_k marks the point/moment in which all components should synchronise to perform the l-th most recent application of rule k.

Definition 4.4. $SO_{i,s} \mathrel{\widehat{=}} Occur(SF_{i,s})$ 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^l_k. Note that this occurrence suffix can be the empty sequence and $l = 0$ denotes the first occurrence. We use $SO_{i,s,j}$ to denote the occurrence variable in position j in $SO_{i,s}$ where $j \in \{1 \dots m_{i,s}\}$ 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 4.1 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 $s = (s_1, \dots, s_n)$, 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 $i \in \{1 \dots n\}$, 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 4.5. Let $\mathcal{S} = (\langle L_1, \dots, L_n \rangle, \mathcal{R})$ 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: $O_{i,s} \mathrel{\widehat{=}} \{SO_{i,s,j} \mid j \in \{1 \dots m_{i,s}\}\}$ gives the set of elements in $SO_{i,s}$, $O \mathrel{\widehat{=}} \bigcup_{i \in \{1 \dots n\}, s \in S'_i} O_{i,s}$ the universal set of occurrences, and

$O_i \mathrel{\widehat{=}} \{o_k^l \mid o_k^l \in O \land r_k \in \Sigma_i'\}$ the set of rule occurrences requiring the participation of component i. Finally, we use $clk_{i,s,j}$ to denote clk_k^l if $SO_{i,s,j} = o_k^l$.

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

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

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

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

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 $SO_{i,s}$:

$$BC(i,s) \mathrel{\widehat{=}} \bigwedge_{o_k^l \in O_i - O_{i,s}} clk_k^l < clk_{i,s,1}$$

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 4.1 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.

Next, we prove that our predicate over-approximates reachability. Since our component-invariant soundly summarises the behaviour of a component and components must synchronise on shared rules to reach system states, it follows that, for any reachable state, components must be able to, in particular, consistently synchronise on the rule occurrences in the occurrence suffixes we derive.

Theorem 4.6. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. For a system state s, $reachable(s) \Rightarrow reach_{\mathcal{S}}(s)$.*

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 $tr \upharpoonright \Sigma'_i$ to reach state s_i, where $tr \upharpoonright \Sigma'_i$ 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 $tr \upharpoonright \Sigma'_i$ respect their ordering in tr given how operator $\upharpoonright$ works.

From Lemma 4.7, we know that SF_{i,s_i} is a suffix of $tr \upharpoonright \Sigma'_i$. 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 4.7. *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 4.3. $SF_{i,s}$ 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 4.8. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, 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 4.3. 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 4.1, 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

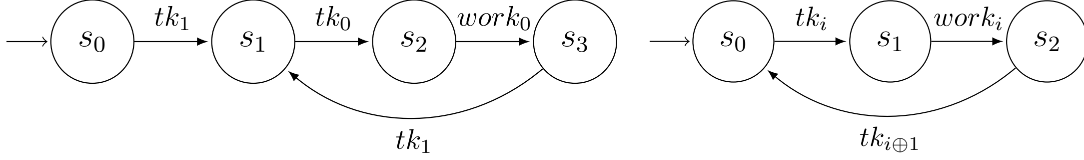

Figure 4.3: LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

system state under analysis are different, we have an upper-bound on occurrence (and clock) variables of $\mathcal{O}(\sum_{i\in\{1...n\}} |L_i|)$, 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 $reach_{\mathcal{S}}(s)$ for a fixed system state s. □

4.2.2.2 Number of rules occurrences consistency

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 4.2 as a running example in introducing this technique.

Example 4.2. This example presents a unidirectional token-ring scheduler; fairly similar to Milner's scheduler. In this system, a token is passed around the ring and the component holding it can work.

This system is described by machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ where L_0, L_1 and L_2 are defined in Figure 4.3 and $\mathcal{R}$ gives 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. 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, 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 4.9. 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;
- $T_{r_h}(\{d\}) \mathrel{\hat{=}} \begin{cases} \{d+1\} & \textbf{if } h = k \\ \{d-1\} & \textbf{if } h = l \\ \{d\} & \textit{otherwise} \end{cases}$
- $T_{r_h}(\emptyset) \mathrel{\hat{=}} \emptyset$ and $T_{r_h}(\mathbb{Z}) \mathrel{\hat{=}} \mathbb{Z}$.

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 4.2, 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 4.10. Let $\mathcal{S} = (\langle L_1, \dots, L_n\rangle, \{r_1, \dots, r_m\})$ 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 $s = (s_1, \dots, s_n)$:

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

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 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 4.2, 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.

The following theorem establishes that $reach_R$ over-approximates reachability. As our difference sets soundly approximate the behaviour of a component and components synchronise on shared rules to reach a system state, 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 4.11. *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 4.14, 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 4.12. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, 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 4.9. The domain that we use in the CSA framework to calculate $DS_i^{k,l}$ has height $h = \mathcal{O}(1)$. So, from Theorem 4.1, 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. □

Difference consistency The second predicate creates a system of equations that simply describes the differences captured by our component-state invariants. This predicate generalises $reach_R$.

Definition 4.13. Let $\mathcal{S} = (\langle L_1, \dots, L_n \rangle, \{r_1, \dots, r_m\})$ 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 $s = (s_1, \dots, s_n)$:

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

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 $reach_R$. The following example shows that $reach_D$ can show unreachability for some system states that $reach_R$ cannot.

Example 4.3. This example presents a system that is very similar to the one introduced in Example 4.2. Here, we also presents an unidirectional token-ring scheduler but instead of a single token, two tokens are passed around. Again, the component holding a token can work.

This system is described by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ where L_0, L_1 and L_2 are defined in Figure 4.4 and $\mathcal{R}$ is 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. 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 L_0 initially.

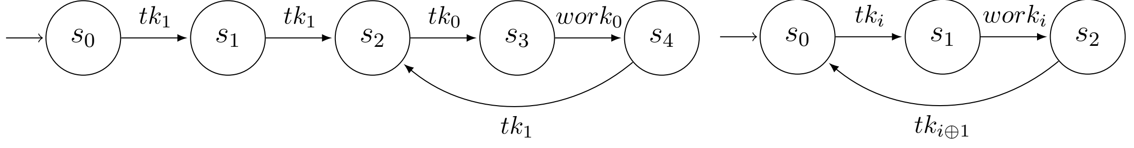

Figure 4.4: LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

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 $DC(0, s_4)$ that (i) $N_{tk_1} - N_{tk_0} = 1$, from $DC(1, s_2)$ that (ii) $N_{tk_1} - N_{tk_2} = 1$, and (iii) $N_{tk_2} - N_{tk_0} = 1$ from $DC(2, s_2)$. So, we can deduce that $N_{tk_1} - N_{tk_0} = 1$ from (i) and $N_{tk_1} - N_{tk_0} = 2$ from (ii) and (iii), 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 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 4.14. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine with induced LTS $(S, \Sigma, \Delta, \hat{s})$. 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 4.15, 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 4.15. *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 4.9. 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 4.16. *Let* $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ *be a supercombinator machine where* $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$*, 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 4.9. The domain that we use in the CSA framework to calculate $DS^{k,l}_i$ has height $h = \mathcal{O}(1)$. So, from Theorem 4.1, 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 of a known fragment of linear integer arithmetic called *difference logic* [KS08]. 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 [Bel58, FF10, CLRS09]

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. □

4.2.3 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 4.3 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 4.13 and RC in Definition 4.10. 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.

4.2.3.1 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. 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 $[r_k]$.

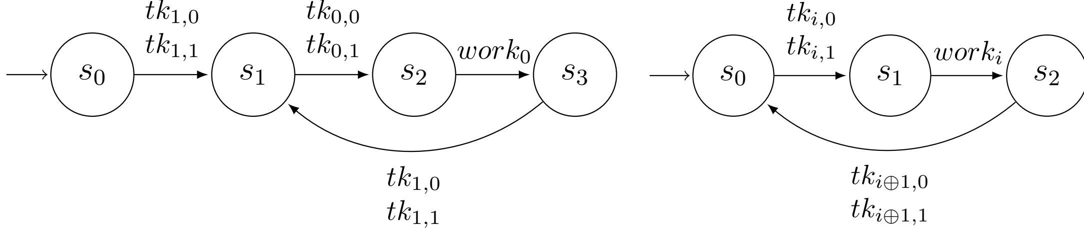

Figure 4.5: LTSs of components L_0 and L_i, for $i \in \{1, 2\}$, respectively.

Definition 4.17. 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 4.4. This example describes again a unidirectional token-ring scheduler. Unlike the previous examples, the token has a value v associated to it and components are free to change this value as they wish. The system has a single token moving around and the component holding it can work.

This system is captured by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 4.5 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. 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 v 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 $N_{[tk_{i,0}]}$ 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 4.2.

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 $reach_R^{DA}((s_3, s_2, s_2))$ ($reach_D^{DA}((s_3, s_2, s_2))$, respectively) 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 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 4.1 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 free, whereas $reach_S$ is unable to capture any useful suffix.

We point out that all levels of abstractions that we propose in this chapter are *incomparable* in the sense that they capture different invariants and, hence, the set of systems they prove 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 4.18. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine with induced LTS $(S, \Sigma, \Delta, \hat{s})$. For $s \in S$, reachable$(s) \Rightarrow reach_x^{DA}(s)$, where x can be S, D, or R.*

4.2.3.2 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 4.19. 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 thesis, 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 4.20. 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.

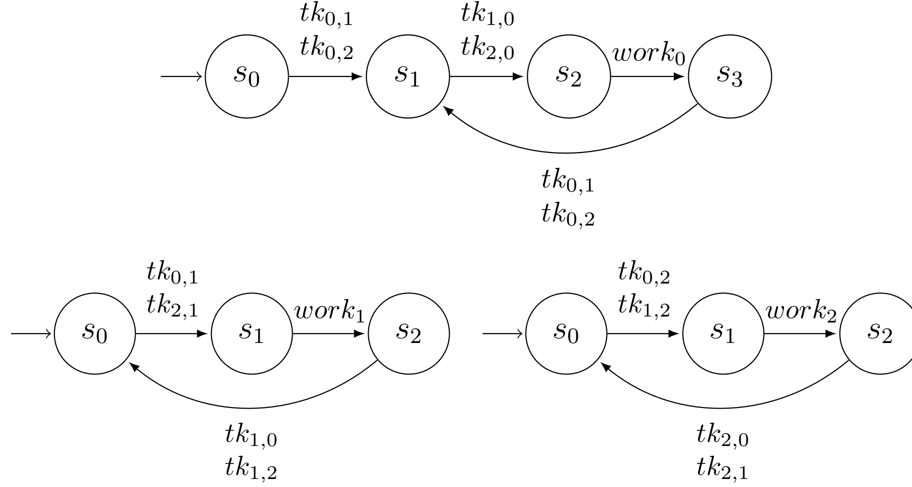

Figure 4.6: LTSs of components L_0, L_1, and L_2, respectively.

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

Example 4.5. This example describes again a token-ring scheduler. Unlike the previous examples, the ring presented here is bidirectional, that is, a component can pass a token either to its predecessor or successor in the ring. The system has a single token moving around (and no value v associated to it) and the component holding it can work.

This system is described by machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 4.6 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. Process L_0 has the token initially and 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: $(N_{tk_{1,0}}+N_{tk_{2,0}})-(N_{tk_{0,2}}+N_{tk_{0,1}}) = 0$, $(N_{tk_{0,1}}+N_{tk_{2,1}})-(N_{tk_{1,0}}+N_{tk_{1,2}}) = 1$, 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 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 4.21. *Let* $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ *be a supercombinator machine with induced LTS* $(S, \Sigma, \Delta, \hat{s})$. *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.

4.2.4 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 the one 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 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 [RD87, Dat89, Mar96, LMC11]. Examples 4.1, 4.2, 4.3, 4.4, and 4.5 all illustrate some common mechanisms that our techniques can capture.

4.3 PairStatic: Pair plus synchronisation analysis

In this section, we propose PairStatic: a framework that improves on Pair by combining 2-reachability with the approximation techniques we propose in this chapter. 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 $PairStatic_x$ *candidate*, where x is R or D depending on the version of our framework.

Definition 4.22. Let $\mathcal{S}$ be a supercombinator machine. A system state s is a $PairStatic_x$ *candidate* iff the following predicate holds, where x can be D or R:

$$candidate_R(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach_S(s) \wedge reach_R(s) \\ \wedge reach_S^{DA}(s) \wedge reach_R^{DA}(s)$$

$$candidate_D(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach_S(s) \wedge reach_D(s) \\ \wedge reach_S^{DA}(s) \wedge reach_D^{DA}(s) \wedge reach_D^{CA}(s)$$

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. The component-specific abstraction requires some arithmetic constraints that are not so straightforward to implement in a SAT solver context. Hence, we decided not to implement a version of $reach_R$ using this abstraction. Note that 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. Table 8.1 (p.225) provides an index-like list of all approximations we propose; it should help the reader throughout this thesis to recapitulate and navigate through the text to find them.

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

Theorem 4.23. *If a supercombinator machine is PairStatic$_x$ candidate free, where x is R or D, then it must also be deadlock free.*

4.3.1 Precision and complexity of PairStatic

This new framework is clearly more precise than Pair, but it remains imprecise: a 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 blocked system state in Example 3.1 is unreachable. So, like Pair, PairStatic is unable to show that the system in that example is 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. Figure 4.7 illustrates the relationship between the precision of (i.e. the set of systems that can be proved deadlock free by) SDD, Pair and PairStatic[2].

[2] Our diagrams are meant to only depict the qualitative relationship of sets. So, we are interested in showing whether sets intersect, a set is contained into another or sets are disjoint; the size of the

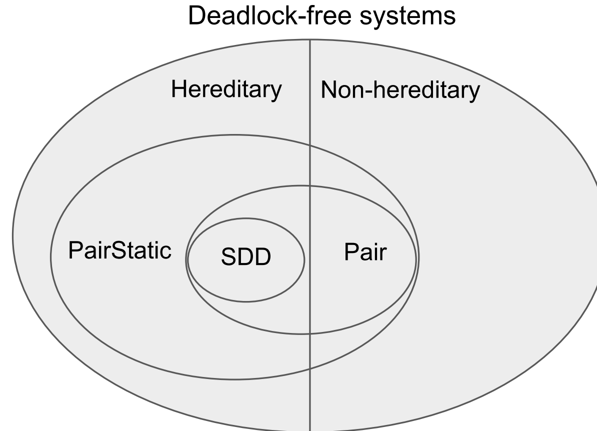

Figure 4.7: The relationship between deadlock-freedom-checking frameworks.

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 4.1, 4.2, 4.3, 4.4 and 4.5 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 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 blocked states and that we can capture with PairStatic. Furthermore, these examples introduce some systems that can be tackled by PairStatic but cannot be shown deadlock free by FSDD or CSDD; we discuss these two techniques in Section 1.1.

The first mechanism we discuss, proposed in [BR91], 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 4.1. In the following, we introduce a modified version of Example 4.1 that can be shown deadlock free by

intersecting areas are immaterial. Thus, for instance, the fact that the PairStatic set lies mostly over the set of hereditary deadlock-free systems does not mean that it can show deadlock freedom for more hereditary deadlock-free systems than non-hereditary deadlock-free systems.

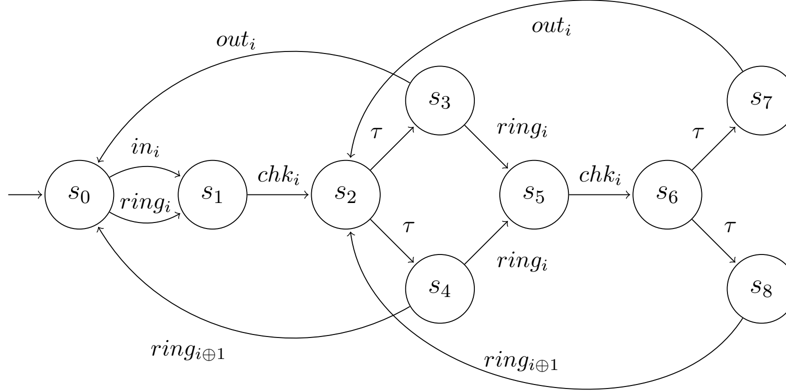

Figure 4.8: LTS of component L_i where $\oplus$ represents addition modulo 3.

PairStatic but not by FSDD. We point out that the work in [Ros87] introduces an approach to systematically transform a non-fillable ring into a deadlock-free system with an arbitrary topology, and PairStatic can capture this sort of invariant even for such an extension.

Example 4.6. This example describes a system that is very similar to the one in Example 4.1. In this ring-like system, components exchange messages unidirectionally around the network. A component receives messages from its predecessor component in the ring or from its user, and it either passes the message over to the next component in the ring or outputs the message to its user. Unlike the system in Example 4.1, once a message is received, its integrity is checked. Each component can hold up to two messages at a time and the second message must have been received from its predecessor and not from its user.

This system is described by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_i defined in Figure 4.8 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 4.1.

Both FSDD and PairStatic can show deadlock freedom for Example 4.1 but only PairStatic can show deadlock freedom for this system. PairStatic can show that this system is deadlock free using a systematic argument that is very similar to the one that we present for Example 4.1. 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 components' last action, $reach_S$ uses

a suffix/sequence of rule events/actions. So, while in Example 4.1 the contradiction arises from components' 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 deadlock freedom for this example.

This example presents a non-fillable system where components store up to two messages. PairStatic, however, can show 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, and as long as the the actions leading to a contradiction can be captured by our suffixes, PairStatic should be able to show deadlock freedom for it. ■

The next mechanism we discuss, proposed in [SD82, RD87], 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 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 illustrate this mechanism. It introduces a system that can be tackled by PairStatic but not by CSDD.

Example 4.7. This example introduces a system that implements a grid-like systolic array. So, components are disposed in a grid-like fashion and each of them behaves cyclically as follows. It interacts with its left and up neighbours, respectively. Then, it does some processing and it, finally, communicates with its right and down neighbours, respectively. Each component initially (non-deterministically) chooses which type of processing it will do.

This system is modelled by supercombinator machine $\mathcal{S} = (\langle L_{1,1}, L_{1,2}, L_{2,1}, L_{2,2}\rangle, \mathcal{R})$ where components $L_{i,j}$ are defined in Figure 4.9 and $\mathcal{R}$ is the set of rules that require components to synchronise on shared events. A component interacts with its left and up neighbours using events $h_{i,j}$ and $v_{i,j}$, respectively. The processing is represented by events $p_{i,j}$ and $p'_{i,j}$, and it communicates with its right and down neighbours via $h_{i+1,j}$ and $v_{i,j+1}$, respectively.

PairStatic can show that this system is deadlock free using a systematic analysis that is very similar to the one that we present for Example 4.2, whereas CSDD cannot. While CSDD captures that a component completed a cycle when it transitions back to

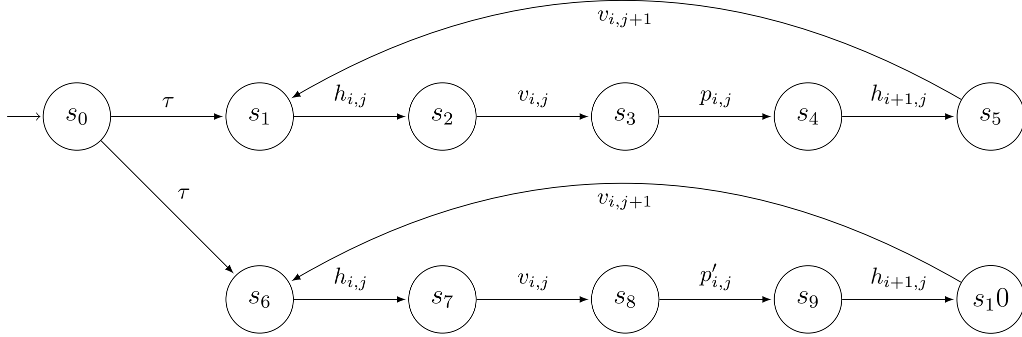

Figure 4.9: Sketch of LTS for components $L_{i,j}$.

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 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. ■

These two mechanisms are also formally presented and discussed in [Mar96] and [Ros98].

Lastly, we discuss a *token-ring* mechanism that allows for tokens to be exchanged between components. In a token-ring system, components are disposed 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 4.2, 4.3 and 4.4. 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 [Ros87] 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 deadlock-free system in Example 4.5.

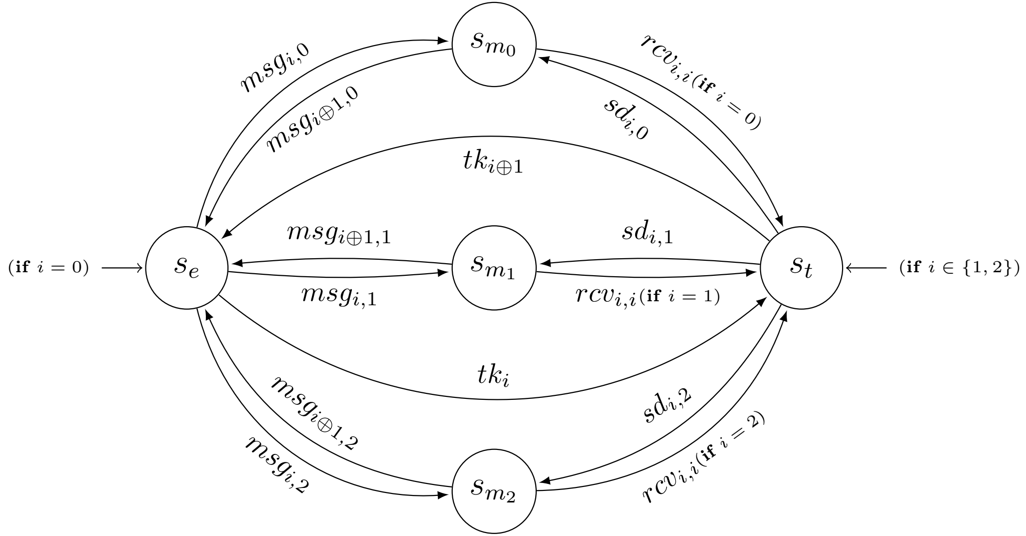

Figure 4.10: Sketch of LTS for components L_i.

Example 4.8. This system implements a token-ring message-exchange system. Components pass tokens around and a component possessing a token can receive a message and relay it around the ring; messages m_i can be exchanged. A component can be empty (state s_e) at which point it can receive a token or a message from other component. If the component holds a token (state s_t), it can receive a message from its user or pass the token around the ring. Once a component has message m_i (state s_{m_i}, depending on the message m_i received), the message can be passed around until eventually outputted by component i to its user. As a result of outputting a message, the component ends up holding a token.

Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_i defined in Figure 4.10 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. Initially, L_1 and L_2 hold tokens 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. Event $msg_{i,v}$ represents the passage of message m_v from $L_{i\ominus 1}$ to L_i. Event $sd_{i,v}$ ($rcv_{i,v}$) depicts the sending (receiving) of message m_v to (from) component i from (to) its user.

Again, PairStatic can show that this system is deadlock free using a systematic analysis that is very similar to the one that we carried out for Example 4.4. 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 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 deadlock free. ■

Generally speaking, our strategy should prove deadlock freedom if 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 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, our combination of approximations allows 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, 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. These results also justify our use of SAT and SMT solving in tackling these problems.

Theorem 4.24. *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.*

Proof. While membership to *NP* follows from the fact that all our predicates can be decided in polynomial time thanks to Lemmas 3.9, 4.8, 4.16, and 4.12, *NP*-hardness follows from Corollary 3.20. □

4.3.2 PairStatic-candidate detection via SAT/SMT solving

We built upon our SAT-checking approach proposed for detecting Pair candidates to create an efficient implementation for PairStatic. We implement our framework using SAT formula $PairStatic_R$ and SMT formula $PairStatic_D$. While $PairStatic_R$ captures states that satisfy $candidate_R$, $PairStatic_D$ detects states satisfying $candidate_D$. We choose to use the theory of linear integer arithmetic to encode $PairStatic_D$ as it is more convenient and SMT solvers tend to be efficient in handling such formulas. So, we encode the search for a 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 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.

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

We reuse the sub-formulas for *State*, *Blocked* and $Reach_2$ defined in Section 3.3.3. Due to the reuse of formula *Blocked*, these encodings can only be applied to triple-disjoint systems (see Section 3.3.3). 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 4.5, for the corresponding predicate[3]. 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.

[3]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*.

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.

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.

4.3.3 Practical evaluation

We extend DeadlOx to implement PairStatic. It constructs our SAT and SMT encodings which are then checked by the Glucose solver and Z3 solver [dMB08], respectively. Our implementation benefits from the incremental nature of modern SAT and SMT solvers to check $PairStatic_R$ and $PairStatic_D$. 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. Our implementation starts solving the conjunction of *State*, *Blocked* and $Reach_2$. If no candidates are found, the system is 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 $PairStatic_R$, the other approximations are conjoined in the following order: $Reach_R^{DA}$, $Reach_S^{DA}$, $Reach_R$ and $Reach_S$. For $PairStatic_D$, 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]]`, which should be added to a deadlock free assertion, to call DeadlOx's $PairStatic_R$ and $PairStatic_D$ frameworks, respectively. DeadlOx's binary and the models used in this section are available at our experiment package [AGRR18].

Our experiment evaluates deadlock freedom for some triple-disjoint deadlock-free systems that cannot be tackled by local analysis alone. Hence, Pair and SDD are unable to show deadlock freedom for all examples discussed in this section. We used

a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. We compare our frameworks $PairStatic_R$ (PS) and $PairStatic_D$ (PSsmt) against Deadlock Checker's FSDD and CSDD frameworks [MJ97], FDR4's methods (FDR, FDRc, FDRp), and D-Finder 2's methods (DF2pm, DF2fp, DF2l).

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 4.1 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) [GS10], 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 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 deadlock freedom for this system. Note that the $PairStatic_D$ encoding can capture the counting mechanism used by But to avoid undesired states and the timestamp-based mechanism implemented by TS, whereas $PairStatic_R$ cannot. The reason is that only the use of our component-specific abstraction presented in Section 4.2.3.2 enables the capture of these mechanisms. The

		Approximate					Exact		
Example	N	FSDD	CSDD	PS	PSsmt	DF2pm	FDR	FDRc	FDRp
But	50	-	-	-	1.07	*	*	*	*
	100	-	-	-	4.87	*	*	*	*
	150	-	-	-	17.39	*	*	*	*
	200	*	*	-	32.52	*	*	*	*
DDB	5	-	-	0.31	0.36	*	0.51	0.16	0.21
	10	*	*	1.27	7.78	*	*	*	*
	15	*	*	10.53	79.41	*	*	*	*
	20	*	*	48.91	*	*	*	*	*
HexSys	3	-	0.20	0.16	0.16	*	*	0.16	0.36
	5	-	0.28	3.57	0.67	*	*	7.33	*
	8	-	0.53	56.12	8.13	*	*	*	*
	10	-	0.68	160.63	19.55	*	*	*	*
Mat	5	-	0.18	0.36	0.16	*	*	0.21	0.17
	10	-	0.23	3.87	0.72	*	*	15.59	0.67
	20	-	0.43	38.84	12.24	*	*	*	28.51
	30	-	0.83	269.55	183.45	*	*	*	*
RingP	5	-	-	0.22	0.16	*	0.26	+	0.16
	10	-	-	0.41	2.32	*	*	+	*
	15	-	-	2.32	24.21	*	*	+	*
	20	*	*	10.43	158.51	*	*	+	*
Ring	100	0.18	-	0.17	0.26	38.44	*	0.72	*
	200	0.28	-	0.26	0.47	*	*	1.57	*
	300	0.38	-	0.42	0.61	*	*	2.72	*
	400	0.43	-	0.51	0.87	*	*	4.22	*
Sched	100	-	0.18	0.11	0.16	5.27	*	0.37	0.41
	200	-	0.28	0.21	0.26	10.18	*	0.72	3.82
	300	-	0.38	0.31	0.31	18.00	*	1.17	15.74
	400	-	0.43	0.47	0.46	47.45	*	1.77	44.68
TS	3	-	-	-	0.11	4.22	0.06	+	0.06
	5	-	-	-	0.36	121.15	0.11	+	0.11
	10	-	-	-	6.78	*	*	+	*
	15	-	-	-	73.05	*	*	+	*
MVote	3	-	-	-	-	-	0.06	0.11	0.11
	5	-	-	-	-	-	0.11	0.31	0.16
	7	-	-	-	-	-	24.46	*	75.55
	10	-	-	-	-	*	*	*	*

Table 4.1: Techniques comparison. 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.

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.

		Approximate					Exact		
Example	N	PS	PSsmt	DF2pm	DF2fp	DF2l	FDR	FDRc	FDRp
MsgGrid	20	-	0.16	15.49	21.50	20.40	*	0.26	*
	40	-	0.21	*	*	*	*	0.77	*
	60	-	0.26	*	*	*	*	1.67	*
	80	-	0.36	*	*	*	*	3.82	*
Ring2	10	0.72	1.47	*	14.69	*	0.37	+	0.46
	15	1.92	14.99	*	*	*	0.46	+	5.27
	20	10.88	79.67	*	*	*	1.07	+	35.87
	25	41.49	*	*	*	*	2.87	+	162.71
Ring2Hf	10	0.31	1.42	*	*	*	*	+	*
	15	1.92	14.24	*	*	*	*	+	*
	20	10.78	83.42	*	*	*	*	+	*
	25	41.29	*	*	*	*	*	+	*
Ring2S	50	1.27	2.17	*	217.61	*	11.60	+	45.03
	100	4.97	22.26	*	*	*	*	+	*
	150	19.86	76.16	*	*	*	*	+	*
	200	55.62	261.42	*	*	*	*	+	*
Ring2SHf	50	1.17	2.62	*	*	*	*	+	*
	100	4.97	22.31	*	*	*	*	+	*
	150	20.00	75.06	*	*	*	*	+	*
	200	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 4.2: Techniques comparison. 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.

Table 4.2 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); 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. $PairStatic_R$ 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 4.1) are predictable enough for $PairStatic_R$ to capture. On the other hand, the routes around the grid network implemented by MsgGrid are too unpredictable to be captured by $PairStatic_R$ but they can still be captured by $PairStatic_D$. Track and TrackHf, however, allow token routes that are too unpredictable even for $PairStatic_D$; the component-specific abstraction implemented in $PairStatic_D$ 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 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 4.3 presents the analysis of token networks implementing three communication topologies: fully-connected, grid, bidirectional ring. The name of our examples describe the topology used. For examples suffixed by *Hf*, components exchange $N/2$ tokens, otherwise they exchange only two. These networks implement token routes that can only be captured by our $PairStatic_D$ 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 4.2 deterministically choose a partner to pass the token to. Here, this choice is made non-deterministically. These results show that $PairStatic_D$, 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 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. These results also demonstrate a shortcoming of our framework. For some systems, deadlock freedom depends on the fact that

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

Table 4.3: Techniques comparison. 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.

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 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.

4.4 Conclusion

This chapter 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 we are aware, our idea of synchronisation analysis and the techniques we implement are new.

We introduce three synchronisation-analysis techniques. The first technique, formalised by predicate $reach_S$, tries to show that a system state is unreachable by demonstrating that components cannot agree on the order they participate in system rules. It examines traces (i.e. suffixes) of rule events that lead each component to its corresponding component state in the system state under analysis. These suffixes capture the most recent behaviour that the components must have performed before reaching this state. So, this technique tries to find inconsistencies between these recent behaviours of components. The second and third techniques, captured by predicates $reach_R$ and $reach_D$, try to establish that a system state is unreachable by demonstrating components cannot agree on the number of times they participate on system rules. Both these techniques try to find meaningful relationships between variables representing the number of times components must participate on system rules to reach a given state; these relationships are checked for consistency. Both the recent behaviour of a component and relationships between number of participations on rule variables can be interpreted as capturing some relevant properties of individual components. The consistency notion between these component properties we employ can, then, be interpreted as capturing system invariants. For instance, the difference in participation on system rules can be interpreted as approximating the token balance of components. In turn, the consistency notion on these relationships can be interpreted as capturing the fact that tokens must be conservatively exchanged between components; a system invariant. We point out that our 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 reachability. 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 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 free systems that are beyond the capabilities of Pair. We present some experimental evidence that suggests PairStatic is able to prove 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.

Our synchronisation-analysis techniques were inspired by Martins's CSDD and FSDD, which were in turn inspired by proof rules from [RD87]. 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 data-flow analysis 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. Other approaches use similar elements of data-flow analysis to approximate reachability [CK94, DCCN04] but they do so in a quite different way and with a rather distinct purpose.

Chapter 5

Global analysis via token structures and invariants

5.1 Introduction

Many concurrent and distributed systems rely on token mechanism to avoid reaching undesired states. In the previous chapter, we introduce techniques that rely on flexible invariants that only incidentally capture token mechanisms. The generality of these invariants makes these techniques unable to detect such mechanisms when token routes are somewhat unpredictable. In this chapter, we address this limitation by proposing techniques that are designed to detect token structures, regardless of how unpredictable they are and even when the system designer has no idea they are there. Based on these structures, they construct global invariants that serve as reachability over-approximations. Hence, they complement the techniques we presented so far. Dealing only with token structures and invariants might seem restrictive but many interaction mechanisms can be interpreted as such.

For systems implementing token mechanisms, understanding and recognising these structures often leads to system invariants (i.e. system abstractions) that are sufficiently strong to prove safety properties. For instance, token invariants are frequently used to show mutual-exclusion properties and deadlock freedom. In this chapter, we propose two techniques that can recognise token structures using SAT and SMT checking. One technique detects token structures where the number of tokens is conserved at all times, whereas the other ensures that at least one token exists in the system at all times. These underlying token invariants (token conservation and existence, respectively) naturally over-approximate reachability. All reachable states must have the same number of tokens according to a conservative token structure, or all reachable states must have at least one token according to an existential structure.

Our techniques try to detect *virtual* tokens, that is, the tokens that we detect are not necessarily a concrete part of the system but an abstract element of the mechanism components use to interact. Our detection techniques simply try to assign to each component state the number of tokens the component holds at that point; this assignment represents a token structure. This assignment must also respect a policy that dictates how components can manipulate tokens. For instance, if we are trying to detect conservative token structures, we must find an assignment that captures that tokens can be exchanged but not created or destroyed. Note how this policy naturally leads to the token conservation invariant discussed.

To demonstrate how these structures can be used in the analysis of safety properties, we combine our detection techniques with Pair to create a more precise, albeit still incomplete, deadlock-freedom-checking framework. Building on our previous frameworks, we encode token invariants into a constraint that is combined with Pair's constraint and later checked by a SAT/SMT solver to yield if the system at hand is deadlock free. This new framework handles a different class of system than current approximate techniques and PairStatic, as it tries to address some of the imprecision that comes with the use of component-state invariants to detect token mechanisms. We extend the DeadlOx tool to implement our framework and detection techniques.

Unlike previous chapters, this one heavily relies on the assumption that systems are triple disjoint. It allows us to construct a simpler and more compact SAT and SMT formula to capture token mechanisms. So, all techniques and frameworks that we propose in this chapter require triple disjointness; otherwise, they are unsound. It should be noted, however, that slightly larger and more complex constraints could be proposed for handling general systems, but we do not further investigate this avenue.

This chapter's outline is as follows. Section 5.2 introduces some reachability approximations that implement global analysis by detecting and deriving token structures and invariants. Section 5.3 introduces PairToken, a framework that combines Pair's local analysis with reachability approximations based on token invariants. Finally, in Section 5.4, we present our concluding remarks.

5.2 Approximate reachability via token structures and invariants

Many concurrent systems use some sort of token mechanism to guide interactions between components and avoid undesired behaviours. In this section, we present two techniques that interpret concurrent systems as token networks, trying to understand

how *virtual* tokens might flow in these systems. We use "virtual" as tokens need not be explicit in the system itself but rather an element of the abstract token mechanism it employs. Each technique assumes a particular policy that specifies how tokens flow. So, our techniques try to assign to each state of each component the number of tokens it holds; this *token structure* represents a token flow. This structure is later used to create reachability invariants (i.e. predicates over system states that over-approximate reachability). We call the process of trying to find this mapping of component states to number of tokens *token-structure detection*. In our approach this detection is done by SAT or SMT solvers. Token structures and invariants tend to emerge from the behaviour of large combinations of components, if not the entire system. Hence, our techniques should complement local analysis by effectively capturing token-based global invariants of systems.

The techniques in this chapter analyse how individual and pair of components behave using the following projections. They differ from the projection employed in the analysis of subsystems in the way they capture system rules. While our subsystem projection might truncate rules, these new projections copy rules that involve exactly the individual or pair of components involved in the projection.

Definition 5.1. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a triple-disjoint supercombinator machine. S_i is the supercombinator machine capturing the projection for component i, whereas $S_{i,j}$ is the one capturing the pairwise projection for components i and j.

- $\mathcal{S}_i = (\langle L_i \rangle, \{((e_i), a) \mid (e, a) \in \mathcal{R} \wedge e_i \neq - \wedge \forall j : \{1 \ldots n\} - \{i\} \bullet e_j = -\})$
- $\mathcal{S}_{i,j} = (\langle L_i, L_j \rangle, \{((e_i, e_j), a) \mid (e, a) \in \mathcal{R} \wedge e_i \neq - \wedge e_j \neq -\})$

Our detection techniques are designed around the assumption that systems are triple disjoint. This assumption enables us to specify token structures based on the analysis of individual and pairs of components. So, we can create SAT and SMT constraints to detect structures that are simpler and more compact. An encoding for more general systems can be devised but would be more complex and less compact.

Each technique proposes a formula $\mathcal{F}$ that is a specification for a type of token structure. This formula is constructed using variables $t_{i,s}$, which capture the number of tokens component i holds in its state s, in a way that a satisfying assignment to variables $t_{i,s}$ forms a valid token structure.

$\mathcal{F}$ is a conjunction of sub-formulas, the most important of which is *Policy*. This sub-formula enforces a token-flow policy; it dictates how tokens can behave when components (inter)act, i.e. a system transition takes place. As the system being

analysed is triple disjoint, either a component acts on its own (i.e. an individual transition takes place) or a pair of components agrees on a rule and interact (i.e. a pairwise transition takes place). So, this sub-formula relies on constraint $enc_i(s, s')$ to dictate how tokens are to be manipulated by individual transitions, whereas $enc_{i,j}(s, s')$ is its counterpart for pairwise transitions.

5.2.1 Binary conservative technique

The first technique we propose, which we refer to as the *binary conservative* technique, tries to capture a binary token structure (i.e. each component always holds either zero or one token) that implements a token-conservation policy (i.e. the number of tokens in the system is invariant). In a binary token structure each component can hold at most one token at a given point. So, $t_{i,s}$ are boolean variables and $\mathcal{F}$ is a boolean formula. The conservation policy is enforced using the following $enc_i(s, s')$ and $enc_{i,j}(s, s')$ constraints.

For an individual transition $(s = (s_i), a, s' = (s'_i))$ involving component i:

$$enc_i(s, s') \mathrel{\widehat{=}} t_{i,s_i} \leftrightarrow t_{i,s'_i} \tag{5.1}$$

For a pairwise transition $(s = (s_i, s_j), a, s' = (s'_i, s'_j))$ of components i and j:

$$\begin{aligned} enc_{i,j}(s, s') \mathrel{\widehat{=}} \; & max_{i,s_i,j,s_j} \leftrightarrow (t_{i,s_i} \vee t_{j,s_j}) \wedge max_{i,s'_i,j,s'_j} \leftrightarrow (t_{i,s'_i} \vee t_{j,s'_j}) \\ & \wedge min_{i,s_i,j,s_j} \leftrightarrow (t_{i,s_i} \wedge t_{j,s_j}) \wedge min_{i,s'_i,j,s'_j} \leftrightarrow (t_{i,s'_i} \wedge t_{j,s'_j}) \\ & \wedge max_{i,s_i,j,s_j} \leftrightarrow max_{i,s'_i,j,s'_j} \wedge min_{i,s_i,j,s_j} \leftrightarrow min_{i,s'_i,j,s'_j} \end{aligned} \tag{5.2}$$

Both these constraints ensure that the sum of tokens held by components at the source state is the same as in the target state of the transition. For a pairwise transition, the source-state sum is captured by auxiliary variables max_{i,s_i,j,s_j} and min_{i,s_i,j,s_j}, whereas the target-state one is captured by max_{i,s'_i,j,s'_j} and min_{i,s'_i,j,s'_j}. These variables capture the unary representations for these sums: the number of $true$ values indicates the value of the sum. For instance, if we have $max_{i,s_i,j,s_j} = true$ and $min_{i,s_i,j,s_j} = true$, the source component states hold 2 tokens; if $max_{i,s_i,j,s_j} = true$ and $min_{i,s_i,j,s_j} = false$, the source component states hold 1 token; and if $max_{i,s_i,j,s_j} = false$ and $min_{i,s_i,j,s_j} = false$, the source component states hold no tokens. The same hold for the primed version of these variables and target states. We ensure that source and target states hold the same number of tokens by enforcing that max_{i,s_i,j,s_j} and min_{i,s_i,j,s_j} have the same value as max'_{i,s_i,j,s_j} and min'_{i,s_i,j,s_j}, respectively. Note that a pairwise transition allows tokens to be passed from i to j or vice versa.

Our *Policy* sub-formula uses these two constraints to enforce that for each system transition $(s = (s_1, \ldots, s_n), a, s' = (s'_1, \ldots, s'_n))$ the number of tokens held by components in s must be the same as s'. Thanks to triple-disjointness, this policy can be enforced with the help of our projections. *Policy* enforces $enc_i((s_i), (s'_i))$ for all transitions $((s_i), a, (s'_i))$ of projection $\mathcal{S}_i$ and $enc_{i,j}((s_i, s_j), (s'_i, s'_j))$ for all transitions $((s_i, s_j), a, (s'_i, s'_j))$ of projection $\mathcal{S}_{i,j}$. Each system transition (s, a, s') is either an individual transition that takes component i from s_i to s'_i or a pairwise transition that takes i and j from s_i and s_j to s'_i to s'_j. In the former case, the system rule enabling this individual transition gives rise to a rule in projection $\mathcal{S}_i$ that allows the transition $((s_i), a, (s'_i))$. Since *Policy* enforces $enc_i((s_i), (s'_i))$, it must be the case that the number of tokens in s and s' are the same. In the latter case, a similar argument can be made using the pairwise projection $\mathcal{S}_{i,j}$ for the pairwise transition taking i and j from s_i and s_j to s'_i to s'_j. Note that we can enumerate the transitions of individual and pairwise projections in polynomial time.

Definition 5.2. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, Δ_i the transition relation of the LTS induced by projection $\mathcal{S}_i$, $\Delta_{i,j}$ the transition relation of the LTS induced by projection $\mathcal{S}_{i,j}$.

$$Policy \mathrel{\widehat{=}} \Big(\bigwedge_{\substack{i \in \{1 \ldots n\} \\ \wedge (s,a,s') \in \Delta_i}} enc_i(s, s') \Big) \wedge \Big(\bigwedge_{\substack{i,j \in \{1 \ldots n\} \wedge i \neq j \\ \wedge (s,a,s') \in \Delta_{i,j}}} enc_{i,j}(s, s') \Big)$$

The triple-disjoint assumption allows us to explicitly find the possible transitions of the system by examining individual and pairs of components. We do not need to indirectly capture transitions by encoding system rules and how they can be fired. Instead, we can explicitly analyse these small parts and directly encode which source states may lead to which target ones. The indirect approach could be used to construct an alternative encoding for *Policy* that would work on general systems. This alternative encoding should still be rather compact but harder to solve.

The other two sub-formulas of $\mathcal{F}$ are $NotAlwaysHoldingToken$ and $Participation$. They forbid some trivial structures (i.e. in which tokens do not get exchanged between components) from being valid assignments for our formula. Sub-formula $NotAlwaysHoldingToken$ forbids assignments where some component always holds a token, though we do permit some components to never hold a token. $Participation$ requires the system to hold at least one token initially. To implement $Participation$, we create the participation variables p_i. In a satisfying assignment, the variable p_i states whether component i participates on the token-flow represented by this assignment. These variables play an important role as we present later.

Definition 5.3. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where S_i and $\hat{s}_i$ gives the set of states and the starting state of component L_i, respectively.

$$NotAlwaysHoldingToken \mathrel{\widehat{=}} \bigwedge_{i \in \{1 \ldots n\}} (\bigvee_{s \in S_i} \neg t_{i,s})$$
$$Participation \mathrel{\widehat{=}} \bigwedge_{i \in \{1 \ldots n\}} (p_i \leftrightarrow (\bigvee_{s \in S_i} t_{i,s})) \wedge \bigvee_{i \in \{1 \ldots n\}} t_{i,\hat{s}_i}$$

So, the binary conservative technique relies on SAT formula $\mathcal{F} \mathrel{\widehat{=}} Policy \wedge NotAlwaysHoldingToken \wedge Participation$, where $Policy$ is constructed using enc_i and $enc_{i,j}$ as defined in (5.1) and (5.2), respectively.

We can simply solve formula $\mathcal{F}$ to find a useful token structure. Nevertheless, we decide to go further and propose an algorithm that uses $\mathcal{F}$ to systematically find different token structures for the system at hand. Function FindStructure in Algorithm 5.1 tries to identify different token structures for different parts of the system; this does not preclude our technique from finding a single token structure for the entire system, if the system only assumes system-wide structures. This algorithm implements a heuristic that tries to localise token structures. By confining tokens to particular small parts of the system, we believe that we can better understand their flow around the system. Next, we give a detailed description of this function.

Given an input supercombinator machine, Function FindStructure in Algorithm 5.1 produces (i.e. detects) a number of tokens structures that are stored in fields *partition* and *structure*. It uses function Solve that returns whether a SAT formula is satisfiable and updates the global field $\mathcal{A}$ with a satisfying assignment. When Solve is called for an unsatisfiable formula, $\mathcal{A}$ is not updated. $\mathcal{A}(var)$ denotes the value assigned to variable var on assignment $\mathcal{A}$.

The call to Solve in FindStructure tries to find a structure for some subsystem of $\mathcal{S}$. Note that the *Participation* clause only requires some subsystem to participate in a token-flow. If a structure is found, it is minimised by Minimise. The minimal structure is, then, recorded by ExtractStructure. We modify our formula at the end of each iteration to ensure that in the next iteration we look for a structure for a different subsystem; this also guarantees that our function terminates.

Given the token structure found by Solve in FindStructure, the function Minimise minimises the input token structure to find a minimal one. That is, it iteratively minimises the subsystem participating in the current token-flow/structure (i.e. the one given by the current satisfying assignment in $\mathcal{A}$), making sure a component in this subsystem holds a token initially, until a minimal subsystem is found. Beginning with the structure found by Solve in FindStructure, at each iteration it tries to find a structure involving a strictly smaller subsystem. Finally, ExtractStructure this

Algorithm 5.1 Algorithm to find conservative token-structures

```
1: function FindStructure(S)
2:     partitions := ∅; structure := ∅
3:     Construct F for S
4:     while Solve(F) do
5:         Minimise(F)
6:         ExtractStructure
7:         F := F ∧ ( ⋀_{i∈{1...n}∧A(p_i)} ¬p_i )
8:     end while
9: end function

10: function Minimise(F)
11:     F' := F
12:     repeat
13:         F' := F' ∧ ( ⋁_{i∈{1...n} ∧A(p_i)} ¬p_i ) ∧ ( ⋁_{i∈{1...n} ∧A(p_i)} t_{i,ŝ_i} ) ∧ ( ⋀_{i∈{1...n} ∧¬A(p_i)} ¬p_i )
14:     until not Solve(F')
15: end function

16: function ExtractStructure
17:     partitions := partitions ∪ {{i | i ∈ {1...n} ∧ A(p_i)}}
18:     structure := structure ∪ {(i, s, A(t_{i,s})) | i ∈ {1...n} ∧ s ∈ S_i ∧ A(p_i)}
19: end function
```

minimal token structure is stored in the global fields *partitions* and *structure*; while the former stores participating subsystem, the latter stores the structures themselves.

The proposed minimisation attempts to more finely capture the behaviour of systems. Small(er) subsystems imply that we know more precisely where tokens are confined, and so, we have a better understanding on how tokens can move around. For instance, we can better identify illegal behaviours such as a token that has moved between two confined subsystems. To be more concrete, let us assume that a system implements two token structures. In the first one, components 1 and 2 share/exchange one token, whereas in the second one, components 3 and 4 share another token. Two iterations of our algorithm could find these two distinct token structures, which can demonstrate that a state where components 1 and 2 both hold a token is unreachable. However, without this iterative-minimising algorithm, a token structure with two tokens being shared amongst these four components would be found. In this latter case, a state where components 1 and 2 both hold a token is no longer deemed unreachable; this sort of false positive is what we are trying to avoid with our algorithm.

We use the information recorded in *partitions* and *structure* to create reachability invariants. As we enforce the preservation of tokens for any system transition, all reachable states must have the same number of tokens. So, we can calculate the number of tokens at the initial state and use it to enforce the following *sum invariant*; we systematically enforce it for each subsystem in *partitions*.

Definition 5.4. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, *partitions* and *structure* the sets recorded after the execution of FINDSTRUCTURE($\mathcal{S}$) in Algorithm 5.1, and $structure(i, s)$ yields 1 if the state s of component i is assigned to true (i.e. $(i, s, true) \in structure$), and 0 otherwise. We create the reachability invariant $reach_{C_{\mathbb{B}}}(s) \mathrel{\widehat{=}} \forall\, sub \in partitions \bullet N(sub) = Tks(sub, s)$, where $N(sub) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, \hat{s}_i)$, and $Tks(sub, s) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, s_i)$.

Lemma 5.5. $reachable(s) \Rightarrow reach_{C_{\mathbb{B}}}(s)$

Proof. Let sub be a subsystem in *partitions*. For $\hat{s}$ the starting state of the LTS induced by system $\mathcal{S}$, we have trivially that $N(sub) = Tks(sub, \hat{s})$. Each transition preserves this invariant, that is, if $(s, a, s') \in \Delta$ then $Tks(sub, s) = Tks(sub, s')$. This is guaranteed by *Policy*. As we require triple disjointness, $(s, a, s') \in \Delta$ is derived either from a individual rule or a pairwise one.

Let (s, a, s') be an individual transition where component i participates. We have that $((s_i), a, (s'_i))$ is a transition of the LTS induced by $\mathcal{S}_i$, so thanks to *Policy*, we have that $structure(i, s_i) = structure(i, s'_i)$. As for all other components j we have $s_j = s'_j$, we end up with $Tks(sub, s) = Tks(sub, s')$.

Let (s, a, s') be a pairwise transition where components i and j participate. We have that $((s_i, s_j), a, (s'_i, s'_j))$ is a transition of the LTS induced by $\mathcal{S}_{i,j}$ so thanks to *Policy*, we have that $structure(i, s_i) + structure(j, s_j) = structure(i, s'_i) + structure(j, s'_j)$. As for all other components k, we have $s_k = s'_k$, we end up with $Tks(sub, s) = Tks(sub, s')$. □

This technique should be particularly useful when applied to systems that implement a token-conservation mechanism to avoid reaching undesired states. We illustrate the application of this technique with the following example.

Example 5.1. This example introduces a token-network scheduler. Components pass around a single token and the component that possesses it can work. Components can pass a token to any other component in the system.

This system is modelled by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 5.1 and $\mathcal{R}$ the set of rules that require components to

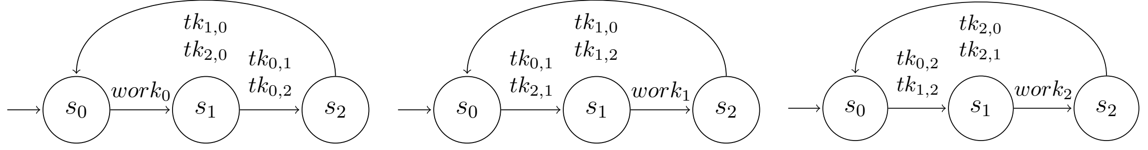

Figure 5.1: LTSs of components L_0, L_1, and L_2, respectively.

synchronise on shared events; e.g. for event $tk_{0,1}$, we have rule $((tk_{0,1}, tk_{0,1}, -), tk_{0,1})$. An arrow with two labels represents two transitions with the same source and target states but with different labels. Process L_0 has the token initially, event $tk_{i,j}$ represents the passage of a token from L_i to L_j, and $work_i$ the work performed by component i.

FINDSTRUCTURE($\mathcal{S}$) can result in *partitions* $= \{\{0, 1, 2\}\}$ and *structure* $= \{(0, s_0), (0, s_1), (1, s_1), (1, s_2), (2, s_1), (2, s_2)\}$[1]; for conciseness, we represent a structure by the states that are assigned to true, so the missing states are assigned to false. With this information, we create the invariant $reach_{C_{\mathbb{B}}}(s) \mathrel{\widehat{=}} Tks(\{0, 1, 2\}, s) = 1$. This invariant can show that this system can never be either filled with tokens, as in (s_1, s_2, s_2), or empty, as in (s_2, s_0, s_0). For instance, (s_1, s_2, s_2) is proved unreachable as $Tks(\{0, 1, 2\}, (s_1, s_2, s_2)) = 3$. As these are the two cases in which this system is blocked, this technique can prove that $\mathcal{S}$ is deadlock-free. $\mathcal{S}$ is a token network with three components and a single token. This technique can, in fact, show that similar systems with N components and n (where $0 < n < N$) tokens are deadlock-free. ■

We point out that if no structure is found, i.e. our specification formula is unsatisfiable, our algorithm computes the empty sets for *partitions* and *structure*. These two empty structures give rise to useless invariants/approximations; our invariants are identified with *true* and all system states are considered reachable. The same happens to all other detection techniques introduced in this chapter.

5.2.2 Binary existential technique

We name our second approach the *existential technique*; components still hold either one or zero token. It enforces a token-flow policy where tokens can be created and destroyed but not eliminated altogether, so there always exists a component holding a token. We implement this new policy using the following definitions for enc_i and $enc_{i,j}$. Constraint $enc_i(s, s')$ allows individual transitions to create but not destroy

[1]It could also result in the token structure given by the complement of *structure*. A formula may admit (be satisfied by) different token structures; we discuss this further in Section 5.2.5.

tokens. For an individual transition $(s = (s_i), a, s' = (s'_i))$ involving component i:

$$enc_i(s, s') \mathrel{\widehat{=}} t_{i,s} \rightarrow t_{i,s'} \tag{5.3}$$

Moreover, $enc_{i,j}(s, s')$ allows components i and j to create or destroy tokens, provided that whenever a token is destroyed one of i and j continues to hold one. Thus the only way a token can be destroyed is in a pairwise transition where both parties hold a token before and only one after. For a pairwise transition $(s = (s_i, s_j), a, s' = (s'_i, s'_j))$ of components i and j, we have the following constraint. We use auxiliary variables has_{i,s_i,js_j} and has_{i,s'_i,j,s'_j} to represent whether a component holds a token in the source s and target s' states, respectively.

$$\begin{aligned} enc_{i,j}(s, s') \mathrel{\widehat{=}}\ & has_{src} \leftrightarrow (t_{i,s_0} \vee t_{j,s_1}) \wedge has_{tgt} \leftrightarrow (t_{i,s'_0} \vee t_{j,s'_1}) \\ & \wedge has_{src} \leftrightarrow has_{tgt} \end{aligned} \tag{5.4}$$

So, this technique uses SAT formula $\mathcal{F} \mathrel{\widehat{=}} Policy \wedge NotAlwaysHoldingToken \wedge Participation$, where $Policy$ is constructed using enc_i and $enc_{i,j}$ as defined in (5.3) and (5.4), respectively.

Again, instead of simply solving formula $\mathcal{F}$ to find a useful token structure, we implement a heuristic to detect (possibly multiple) localised token structures using function FindStructure presented in Algorithm 5.2. This function works similarly to the one presented for the conservative technique, it does, however, a second kind of localisation. Not only we want tokens to be confined to small subsystems, but we want components to hold tokens for as little time as possible; we propose a second minimisation step for this purpose. As for our previous algorithm, this function takes a supercombinator machine as an input and produces a number of localised token structures, which are store in global fields *partitions* and *structure*. The functions Minimise and ExtractStructure are as described in Algorithm 5.1.

While Minimise tries to minimise the subsystem participating in a given structure, FurtherMinimise tries to minimise the timespan in which components hold a token. It works on a token structure that is minimal in terms of participants, which is found by Minimise, and it tries to minimise the number of variables $t_{i,s}$ set to *true*[2]. This second minimisation is an attempt to prevent the creation of spurious tokens; for instance, the creation of unnecessary tokens by individual transitions. Again, structures and participating subsystems are recorded in the global fields *structure* and *partitions*.

[2]Setting the polarity of SAT variables, so that the solver first decides to assign variables to *false*, can substantially speed this minimisation process.

Algorithm 5.2 Algorithm to find existential token-structures

```
1: function FindStructure(S)
2:     partitions := ∅; structure := ∅
3:     Construct F for S
4:     while Solve(F) do
5:         Minimise(F)
6:         FurtherMinimise(F)
7:         ExtractStructure
8:         F := F ∧ ( ⋀_{i∈{1...n}∧A(p_i)} ¬p_i )
9:     end while
10: end function

11: function FurtherMinimise(F)
12:     F'' := F
13:     repeat
14:         F'' := F'' ∧ ( ⋁_{i∈{1...n}∧s∈S_i ∧A(t_{i,s})} ¬t_{i,s} ) ∧ ( ⋀_{i∈{1...n}∧s∈S_i ∧¬A(t_{i,s})} ¬t_{i,s} )
15:     until not Solve(F'')
16: end function
```

The information in *partitions* and *structure* is, once again, used to create reachability invariants. Note that our token-flow policy allows tokens to be destroyed as long as tokens are not completely annihilated from the system. So, as this technique guarantees that at least one token exists initially, a token exists at all times. The reachability invariant that we propose enforce this *existential property* for each subsystem in *partitions*.

Definition 5.6. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, *partitions* and *structure* the sets recorded after the execution of FindStructure($\mathcal{S}$) in Algorithm 5.2, and $structure(i, s)$ yields 1 if the state s of component i is assigned to true, and 0 otherwise. Also, $Tks(sub, s) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, s_i)$. We propose the reachability invariant $reach_{E_{\mathbb{B}}}(s)$ defined as $\forall\, sub \in partitions \bullet Tks(sub, s) \geq 1$.

Lemma 5.7. $reachable(s) \Rightarrow reach_{E_{\mathbb{B}}}(s)$

Proof. Let *sub* be a subsystem in *partitions*. For $\hat{s}$ the starting state of the LTS induced by system $\mathcal{S}$, we have by construction that $Tks(sub, \hat{s}) \geq 1$; Minimise ensures that at least one component in *sub* holds a token. Each transition preserves this invariant, that is, if $(s, a, s') \in \Delta$ and $Tks(sub, s) \geq 1$ then $Tks(sub, s') \geq 1$.

This is guaranteed by *Policy* as follows. Again, triple disjointness guarantees that $(s, a, s') \in \Delta$ is derived either from a individual rule or a pairwise one.

Let (s, a, s') be an individual transition where component i participates. We have that $((s_i), a, (s'_i))$ is a transition of the LTS induced by $\mathcal{S}_i$, so thanks to *Policy*, we have that $structure(i, s'_i) \geq structure(i, s_i)$. As for all other components j we have $s_j = s'_j$, we end up with $Tks(sub, s') \geq Tks(sub, s)$. From $Tks(sub, s') \geq Tks(sub, s)$ and $Tks(sub, s) \geq 1$, we can conclude $Tks(sub, s') \geq 1$.

Let (s, a, s') be a pairwise transition where components i and j participate. We have that $((s_i, s_j), a, (s'_i, s'_j))$ is a transition of the LTS induced by $\mathcal{S}_{i,j}$ so thanks to *Policy*, we have that if $structure(i, s_i) + structure(j, s_j) \geq 1$ then $structure(i, s'_i) + structure(j, s'_j) \geq 1$. Assuming $structure(i, s_i) + structure(j, s_j) \geq 1$, we have $Tks(sub, s') \geq 1$. On the other hand, if $structure(i, s_i) + structure(j, s_j) = 0$, as we have $Tks(sub, s) \geq 1$, there must be a component $k \in sub$ such that $structure(k, s_k) = 1$. Hence, as for all other components k we have $s_k = s'_k$, we have $Tks(sub, s') \geq 1$. □

This technique should be particularly useful when applied to systems where tokens represent a property of components and the fact that at least one component always has this property (i.e. a token) prevents the system from reaching a "bad" state. We illustrate the application of this technique with the following example.

Example 5.2. This example presents a load-balancer system. A load-balancer component activates worker components by giving them a token; the load-balancer component can activate any number of worker components. It can be in an active state, passing tokens to workers, or it can be in an idle state where it waits for a token from any of the workers. When a worker receives a token, it can work and pass the token back to the load-balancer component.

This system is captured by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 5.2 and $\mathcal{R}$ the set of rules that allow components L_1 and L_2 to run independently from each other (no need for mutual synchronisation) but they need to synchronise on shared events with L_0. An arrow with multiple labels is a shorthand for several transitions with the same source and target states but different labels. Component L_0 is the load balancer, and L_1 and L_2 are worker components. Event tk_i captures the sending of a token from to L_i, and $work_i$ the work of component i. This system is constructed so that at least one component is always active (i.e. holding a token) and active components can make the system progress.

By applying FindStructure to $\mathcal{S}$, we can have $partitions = \{\{0, 1, 2\}\}$ and $structure = \{(0, s_0), (1, s_1), (1, s_2), (2, s_1), (2, s_2)\}$. With this information, we create

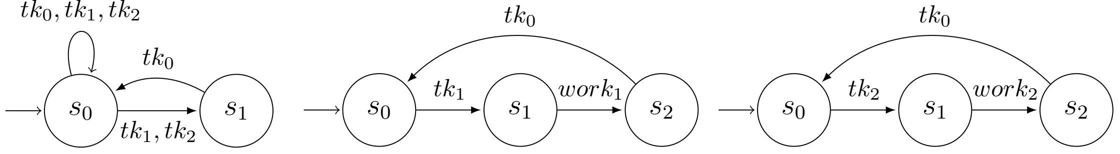

Figure 5.2: LTSs of components L_0, L_1, and L_2, respectively.

invariant $reach_{E_{\mathcal{B}}}(s) \mathrel{\widehat{=}} Tks(\{0,1,2\}, s) \geq 1$; again states assigned to *false* (i.e. with zero tokens) are omitted. This structure pinpoints when components are active and our invariant captures that the system cannot reach a state in which all component are inactive. So, our technique, thanks to this invariant, can show that state (s_1, s_0, s_0) is unreachable; note $Tks(\{0,1,2\}, (s_1, s_0, s_0)) = 0$. As this state is the only state in which the system is blocked, this technique can prove that $\mathcal{S}$ is deadlock-free. $\mathcal{S}$ is a system with two workers. This technique can, in fact, show that similar systems with $N \geq 3$ workers are deadlock-free. ■

Note that the conditions in our specification of a binary conservative structure imply those of a binary existential structure. So, a system that has a binary conservative invariant must have a existential one as well. This observation should be considered when proposing a verification framework built upon these techniques. It seems sensible to, first, detect conservative structures and then, in case the invariants derived from these structures are not strong enough to prove the desired property, we would search for existential structures.

5.2.3 Counter-example-guided conservative technique

In this section, we propose a technique that detects conservative token structures where the number of tokens a component can hold is given by a natural number. This generalisation is built around a SMT formula that uses linear integer arithmetic as its ground theory and where variables $t_{i,s}$ represent natural numbers (i.e. $t_{i,s}$ are integer variables such that $t_{i,s} \geq 0$). This generalisation, which we refer to as the *conservative* technique, uses the following $enc_i(s, s')$ and $enc_{i,j}(s, s')$ constraints to implement its conservative policy.

For an individual transition $(s = (s_i), a, s' = (s'_i))$ involving component i:

$$enc_i(s, s') \mathrel{\widehat{=}} t_{i,s_i} = t_{i,s'_i} \tag{5.5}$$

For a pairwise transition $(s = (s_i, s_j), a, s' = (s'_i, s'_j))$ of components i and j:

$$enc_{i,j}(s, s') \mathrel{\widehat{=}} t_{i,s_i} + t_{j,s_j} = t_{i,s'_i} + t_{j,s'_j} \tag{5.6}$$

Policy implements a token-conservation policy using these two constraints. The original sub-formulas *NotAlwaysHoldingToken* and *Participation*, however, do not make any sense in this new setting. So, we need some new constraints to forbid trivial (and useless) token structures. We propose, then, constraint *InitialToken* and a new *Participation* constraint. *InitialToken* requires the system to hold at least one token initially, whereas *Participation* ensures that at least two components participate on the token structure we are trying to detect. It uses the boolean participation variable p_i to capture whether component i participates in some token manipulation, namely, whether there exists some transition of this component for which there is a change on the number of tokens the component holds.

Definition 5.8. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ gives the initial state of L_i, and Δ_i the transition relation of the LTS induced by projection $\mathcal{S}_i$.

$$
\begin{aligned}
\mathit{InitialToken} \mathrel{\widehat{=}} & \sum_{i \in \{1 \ldots n\}} t_{i,\hat{s}_i} > 0 \\
\mathit{Participation} \mathrel{\widehat{=}} & \bigwedge_{i \in \{1 \ldots n\}} (p_i \leftrightarrow \bigvee_{((s_i),a,(s'_i)) \in \Delta_i} t_{i,s_i} \neq t_{i,s'_i}) \wedge \bigvee_{i \in \{1 \ldots n\}} p_i
\end{aligned}
$$

For this technique, we propose a different method to find token structures. We could use the same approach we propose for the binary conservative technique, namely, we could use FindStructure in Algorithm 5.1, calling a SMT solver instead of a SAT solver, with our new SMT formula[3]. We propose, instead, a simple way to carry out token-structure detection guided by a counter-example. In some of the examples that we tested, we noticed that our original approach would find a few token structures but not the one needed to prove the property at hand. This counter-example-guided detection, however, looks for token structures that prove some bad state unreachable; it is a property-driven detection. We point out that this counter-example-guided technique only detects a single token structure for the system at hand, as opposed to the previous techniques that can find multiple ones.

For a given counter-example system state $ce = (ce_1, \ldots, ce_n)$, where ce is a parameter for this technique, we propose a sub-formula that specifies that the structure to be detected needs to eliminate ce, i.e. it needs to generate a sum invariant that shows that this state is unreachable. In Section 5.3.3, we explain how this counter-example is derived by the incremental solving we propose.

$$
\mathit{EliminateCE} \mathrel{\widehat{=}} \sum_{i \in \{1 \ldots n\}} t_{i,\hat{s}_i} \neq \sum_{i \in \{1 \ldots n\}} t_{i,ce_i}
$$

[3]We would need to replace $(\bigvee_{i \in \{1 \ldots n\} \wedge \mathcal{A}(p_i)} t_{i,\hat{s}_i})$ by $(\sum_{i \in \{1 \ldots n\} \wedge \mathcal{A}(p_i)} t_{i,\hat{s}_i} > 0)$ in updating $\mathcal{F}$ in Minimise.

Algorithm 5.3 Algorithm to find conservative token-structures

1: **function** FINDSTRUCTURE($\mathcal{S}, ce$)
2: $\quad$ $partitions := \emptyset$; $structure := \emptyset$
3: $\quad$ Construct $\mathcal{F}$ for $\mathcal{S}$ and ce
4: $\quad$ **if** SOLVE($\mathcal{F}$) **then**
5: $\quad\quad$ MINIMISE($\mathcal{F}$)
6: $\quad\quad$ EXTRACTSTRUCTURE
7: $\quad$ **end if**
8: **end function**

Then, we have $\mathcal{F} \mathrel{\widehat{=}} Policy \wedge InitialToken \wedge Participation \wedge EliminateCE$, where $Policy$ uses enc_i and $enc_{i,j}$ as defined in (5.5) and (5.6), respectively.

Our conservative technique relies on FINDSTRUCTURE in Algorithm 5.3 to detect token structures. In this function, SOLVE calls a SMT solver instead of a SAT solver, and MINIMISE and EXTRACTSTRUCTURE are as described in Algorithm 5.1. Instead of looking for token structures for different partitions, it only looks for one conservative structure that eliminates counter-example state ce.

This technique relies on the same sum invariant that we propose for the binary conservative technique. Note, however, that token structures need not be binary.

Definition 5.9. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, $partitions$ and $structure$ the sets recorded after the execution of FINDSTRUCTURE($\mathcal{S}, ce$) in Algorithm 5.3, and $structure(i, s)$ yields the number of tokens associated to state s of component i. The reachability invariant $reach^{ce}_{C}(s)$ is as follows:

$$reach^{ce}_{C}(s) \mathrel{\widehat{=}} \forall\, sub \in partitions \bullet N(sub) = Tks(sub, s)$$

where $N(sub) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, \hat{s}_i)$, and $Tks(sub, s) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, s_i)$.

Lemma 5.10. $reachable(s) \Rightarrow reach^{ce}_{C}(s)$

Proof. The proof is very similar to the one for Lemma 5.5. □

This technique extends the capabilities of our binary conservative technique as it can capture structures where a component can hold multiple tokens at once.

5.2.4 Counter-example-guided existential technique

We could also propose a generalised existential technique where components might hold a non-negative integer number of tokens. This generalisation, however, would not lead

to a technique with better precision; a generalised structure can be transformed into a binary one that is (at least) just as precise in terms of the reachability approximation they give rise to.

Definition 5.11. A generalised existential token structure is given by a subsystem *sub* and a mapping *structure* of component states to non-negative integers such that:

1. It respects the existential policy:
   - For pairwise transitions of component i and j from s_i,s_j to s'_i,s'_j,

     $str(i, s_i) + str(j, s_j) > 0 \Leftrightarrow str(i, s'_i) + str(j, s'_j) > 0$
   - For individual transitions of component i from s_i to s'_i,

     $str(i, s_i) > 0 \Rightarrow str(i, s'_i) > 0$

   where $str(i, s)$ is a shorthand for $structure(i, s)$.
2. It has an initial token: $\sum_{i \in sub} str(i, \hat{s}_i) \geq 1$, for initial system state $(\hat{s}_1, \ldots, \hat{s}_n)$.

For the sake of presentation, in the context of this definition and the following theorem, we use relaxed definitions for our (binary and generalised) token structures; we disregard the need for components to participate on a flow and for them not to always hold a token. We could easily strengthen this generalised definition to take into account these other requirements but this would just clutter the point we are trying to make.

Theorem 5.12. *For any generalised existential token structure* $(sub, structure)$ *of the input system, there exists a corresponding binary token structure* $(sub, structure')$ *that proves (at least) the same system states unreachable, that is, for any system state* $(s_1, \ldots, s_n)$*, if* $\sum_{i \in sub} str(i, s_i) = 0$ *then* $\sum_{i \in sub} str'(i, s_i) = 0$.

Proof. Let $structure'$ be given by: $structure'(i, s) = false$ if $structure(i, s) = 0$, and $structure'(i, s) = true$ if $structure(i, s) > 0$. Based on our definition of $structure'$ and requirements 1 and 2 of $(sub, structure)$, it follows that the pair $(sub, structure')$ is a binary existential token structure, namely, it satisfies the existential-token-flow policy and has a token initially. Furthermore, assuming that $\sum_{i \in sub} str(i, s_i) = 0$ and based on our definition of $structure'$, it must be the case that $\sum_{i \in sub} str'(i, s_i) = 0$. □

So, in this section, we only adapt the binary existential technique presented in Section 5.2.2 to carry out counter-example-guided detection of token structures. To do so, we only need to slightly adapt its original formula and algorithm.

Algorithm 5.4 Algorithm to find existential token-structures

```
1: function FindStructure(S, ce)
2:     partitions := ∅; structure := ∅
3:     Construct F for S and ce
4:     if Solve(F) then
5:         Minimise(F)
6:         FurtherMinimise(F)
7:         ExtractStructure
8:     end if
9: end function
```

This technique adds sub-formula $EliminateCE$ to the binary existential technique's original formula. For a given counter-example system state $ce = (ce_1, \ldots, ce_n)$, where ce is a parameter for this technique, it guides the detection procedure to find a structure that eliminates ce, i.e. it needs to generate an existential invariant that proves ce unreachable.

$$EliminateCE \mathrel{\widehat{=}} \bigwedge_{i \in \{1 \ldots n\}} \neg t_{i,ce_i}$$

So, we have formula $\mathcal{F} \mathrel{\widehat{=}} Policy \wedge NotAlwaysHoldingToken \wedge Participation \wedge EliminateCE$, where $Policy$, $NotAlwaysHoldingToken$ and $Participation$ are as described for the original binary existential technique in Section 5.2.2.

To detect structures, our technique relies on FindStructure in Algorithm 5.4. Functions Minimise, FurtherMinimise and ExtractStructure are as described in Algorithms 5.1 and 5.2. Unlike our original technique that looks for structures for different parts of the system, it looks for one existential structure that eliminates counter-example state ce.

This technique relies on the same existential invariant that we propose for the original binary existential technique.

Definition 5.13. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, $partitions$ and $structure$ the sets recorded after the execution of FindStructure($\mathcal{S}, ce$) in Algorithm 5.4, and $structure(i, s)$ yields 1 if the state s of component i is assigned to true, and 0 otherwise. Also, $Tks(sub, s) \mathrel{\widehat{=}} \sum_{i \in sub} structure(i, s_i)$. We propose the following reachability invariant:

$$reach^{ce}_{E_{\mathbb{B}}}(s) \text{ defined as } \forall\, sub \in partitions \bullet Tks(sub, s) \geq 1$$

Lemma 5.14. $reachable(s) \Rightarrow reach^{ce}_{E_{\mathbb{B}}}(s)$

Proof. The proof is very similar to the one for Lemma 5.7. □

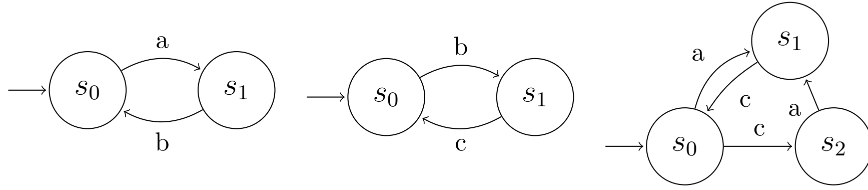

Figure 5.3: LTSs of components L_0, L_1 and L_2, respectively.

5.2.5 Discussion

One drawback of the techniques introduced in this chapter is their unpredictability: a system may admit different token structures and, consequently, different reachability invariants for the same system, and we might not know a priori which structure and invariant will be found. The formulas we use to capture token structures are mere specifications for them; multiple structures might satisfy the formula that we build for a given system. This unpredictability carries over to verification frameworks using these techniques. An undesired effect is that these frameworks might yield different outcomes when verifying the same system. For instance, two different runs of such a framework might find two token structures: one shows that all violating system states are unreachable, whereas the other does not. Both the minimisation procedures we use and the counter-example guided detection proposed should reduce the space of possible token structures for a system, thereby reducing their unpredictability.

We illustrate the unpredictability of our techniques with the following example.

Example 5.3. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine such that L_0, L_1 and L_2 are described in Figure 5.3 and $\mathcal{R}$ requires components to synchronise on shared events. Let us consider $\mathit{partitions} = \{1, 2, 3\}$, $\mathit{structure} = \{(0, s_1), (1, s_1), (2, s_0), (2, s_2)\}$ and $\mathit{structure}' = \{(0, s_0), (1, s_0), (2, s_0), (2, s_1)\}$. For $\mathcal{S}$, the conservative techniques cannot find any structures, while the existential ones might compute either *partition* and *structure* or *partition* and *structure*′. If it computes *structure*, then (s_0, s_0, s_1) is proved unreachable but not (s_1, s_1, s_2). In case *structure*′ is computed, (s_1, s_1, s_2) is proved unreachable but not (s_0, s_0, s_1). As it cannot use *structure* and *structure*′ simultaneously, either (s_1, s_1, s_2) or (s_0, s_0, s_1) will be assumed reachable. Since these are the two system states where the system is blocked, it cannot show that $\mathcal{S}$ is deadlock free. ■

We could modify our techniques to capture several token structures and invariants. In cases where only a handful of structures are needed to prove a property, such as the previous example, this modification would lead to an effective framework. In general, however, a large number of structures might be needed to disprove all violating

states. In these cases, finding all structures would be impractical. This example also demonstrates the imprecise nature of our techniques.

Another source of imprecision is the way our techniques abstract the system's behaviour by the token mechanisms it implements. Therefore, they can only prove properties that emerge from the behaviour of the system's underlying token mechanisms. For instance, if a system implements no such mechanism, our techniques will not be able to detect a token structure and so they will create the useless approximation that assumes every system state reachable. Or, it might even be the case a token mechanism is implemented but this mechanism is not responsible for leading the system away from bad states, so our techniques would not guarantee this property.

That said, our invariants and the approximations they lead to are precise enough to show some interesting properties of concurrent systems. Token mechanisms are commonly implemented by concurrent systems to avoid reaching bad states. For instance, Examples 5.1 and 5.2 illustrate how some well-coordinated protocol for token exchange can ensure that some component is always able to make the system progress, thereby avoiding deadlocks. Moreover, it is fairly common to have token mechanisms being used to guarantee mutual-exclusion properties [Mur89, Ros98]. These techniques were conceived for the analysis of systems implementing obvious token mechanisms. However, many other interaction paradigms of concurrent systems can be interpreted as such. For instance, a system that has a semaphore-like guard component to prevent bad states from being reached, such as the butler solution to the dining philosophers problem, can be seen as implementing a token mechanism of sorts. Also, the non-fillable mechanism introduced in [Dat89, BR91] can also be interpreted as a token mechanism. So, our approximations can also prove properties emerging from the behaviour of these mechanisms. It turns out that even systolic-array-like systems, which we did not anticipate as implementing such a mechanism, admit some token structures, albeit the ones we found were not strong enough to prove the properties we were interested at. We discuss in more details these interaction mechanisms and how they can be captured by our techniques in Section 5.3.1.

The wide applicability of our techniques comes from the generality of our detection mechanisms. We conceived these techniques to find invariants of systems that obviously implement token mechanisms, namely, they use messages (events like tk_i) to move tokens around the system, like Examples 5.1 and 5.2. Note, however, that the notion of a "token" is a mere abstraction; it does not really exist. Our formalism does not explicitly define the notion of a token, neither systems specify special components to denote tokens. Tokens are rather (virtual) abstract elements of the

interaction mechanism systems implement. Since our techniques simply interpret system transitions as some abstract token movement, even though they might not be so, we are able to find token structures and invariants for a variety of systems, even when they do not obviously implement a token mechanism.

5.3 PairToken: Pair meets token invariants

In this section we combine Pair with token invariants. In this new framework, which we call *PairToken*, a potential deadlock is a Pair candidate that meets our new reachability invariants. We propose two different characterisations for a deadlock candidate. The first one relies on our original binary techniques, whereas the second on their counter-example-guided counterparts. Note that the $reach^{ce}_{C}$ and $reach^{ce}_{E_{\mathbb{B}}}$ need not be constructed based on the same counter-example.

Definition 5.15. Let $\mathcal{S}$ be a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $ce, ce' \in S$ two counter-example states. A state $s \in S$ is a *deadlock candidate* iff:

$$candidate_{\mathbb{B}}(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach_{C_{\mathbb{B}}}(s) \wedge reach_{E_{\mathbb{B}}}(s)$$

$$candidate_{ceg}(s) \mathrel{\widehat{=}} blocked(s) \wedge reach_2(s) \wedge reach^{ce}_{C}(s) \wedge reach^{ce'}_{E_{\mathbb{B}}}(s)$$

Since our reachability tests over-approximate reachability, every deadlock must also be a deadlock candidate. So, a system free of deadlock candidates has to be deadlock free.

Theorem 5.16. *If a supercombinator machine is deadlock-candidate free, then it must also be deadlock free.*

Proof. This follows from Lemmas 3.8, 5.5, 5.7, 5.10 and 5.14. □

5.3.1 Precision and complexity of PairToken

Our new deadlock-candidate characterisation is clearly more precise than the one Pair uses. PairToken improves on the precision of Pair by capturing global invariants. Token mechanisms are often implemented over (i.e. with the participation of) all components in the system. So, understanding how tokens flow around the system leads to a system-wide invariant. It remains, however, imprecise: a blocked state can be unreachable and yet meet our reachability invariants. Note that this framework is only imprecise in terms of reachability; a candidate deadlock might not be reachable but it is always blocked. Nevertheless, by conjoining these new reachability tests,

we tighten the state space analysed; it only takes one failed test to consider a state unreachable. Furthermore, we reinforce that our token-detection techniques can cause PairToken to be unpredictable, as illustrated by Example 5.3. Nevertheless, they can capture a number of common interaction mechanisms, some of which maintain global invariants that prevent deadlocks.

Our techniques were designed to verify systems that implement (obvious) token mechanisms by capturing the underlying invariant these mechanisms maintain. For instance, Example 5.1 illustrates how our techniques can detect and use the invariant that all system states must have a single token to show that blocked states, which in this case have either zero or three tokens, are unreachable. On the other hand, Example 5.2 illustrates how our techniques detect an existential structure where tokens denote whether components are active and use the invariant that at least one component must be active at all times to prove deadlock freedom.

Given the generality of our detection techniques, however, it can also detect (interpret) other interaction mechanisms using token structures and invariants. For instance, our conservative techniques verify deadlock freedom for some systems implementing a semaphore-like guard component that leads the system away from deadlocks. Semaphores are very common synchronisation primitives of concurrent systems that are normally used to keep track of the number of components at some specified execution state and ensure that (i) at most T components can be at this state at any given time [Dij68, AS83]. They are implemented as non-negative integer variables that can be incremented up to T and decremented down to zero; incrementing or decrementing is only successful if the resulting value remains within this interval. Components increment this variable before entering the specified execution state and decrement it after they have left it to ensure the counting invariant given by (i). From a token-mechanism perspective, we can interpret the incrementing (decrementing) of the semaphore-like component as the sending (receiving) of a token to (from) it. So, initially this component has T tokens and the components taking part in the incrementing/decrementing process have zero; these tokens are exchanged as increments/decrements are performed. Note that the number of tokens in the system is conserved at all times. The traditional butler solution to the dining philosopher problem [Ros10] is an example of this sort of system.

Example 5.4. Let $\mathcal{S} = (\langle B, P_0, P_1, P_2, F_0, F_1, F_2 \rangle, \mathcal{R})$ be the supercombinator machine with B, F_i and P_i defined in Figures 5.4 and 5.5, and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. We use $\oplus$ and $\ominus$ to denote addition and subtraction modulo 3. $\mathcal{S}$ implements the traditional butler solution to

the dining philosophers problem for three philosophers and forks. In this system, a philosopher P_i sits at a round table (event sit_i) where it has to acquire fork F_i on its left-hand side (event get_i) and fork $F_{i\oplus 1}$ on its right-hand side (event $get_{i\oplus 1}$). With this pair of forks at hand, it can eat (event eat_i), after which it places the forks back at the table (events put_i and $put_{i\oplus 1}$) and then leaves the table (event up_i). A fork F_i is shared between philosophers P_i and $P_{i\ominus 1}$; it represents a resource that can only be acquired and then released by one of these philosophers. If each philosopher is at a point where the left-hand-side fork has been acquired, they are all blocked waiting for their right-hand-side fork to become available. To avoid this state, a butler component B is introduced. It prevents all philosophers from being sat at the table simultaneously, thereby preventing this deadlocked state.

The butler is a semaphore-like component that allows at most two philosophers to be sat at the table at any given time. Taking the semaphore perspective, we can interpret that the different states of the butler component count the number of philosophers sat at the table. When in s_i, i-many philosophers are sat at the table. Also, events sit_i are used by philosopher P_i to increment the semaphore, while up_i decrements it. Note that when in s_2, the semaphore (butler) cannot be incremented, thereby precluding all (three) philosophers from being sat simultaneously. By the token-mechanism perspective, we can interpret that initially the butler has $T = 2$ tokens to be passed around, whereas all philosophers have initially none. Then, events sit_i (up_i) are used to pass a token from (to) the butler to (from) philosopher P_i. Note that, under this perspective, we have a conservative token mechanism.

So, our counter-example-guided conservative technique, where the input counter-example is a state where all philosophers have acquired their left-hand-side fork, could find $partitions = \{\{0, 1, 2, 3\}\}$ and $structure$ would have for butler states the assignments $(s_0, 2), (s_1, 1), (s_2, 0)$, and for each philosopher P_i the state mapping $(s_0, 0)$ and $(s_j, 1)$ for all other states s_j where $j \neq 0$; forks do not participate on this token structure. Thus, we have invariant/approximation $reach^{ce}_{C}(s) \mathrel{\widehat{=}} Tks(\{0, 1, 2, 3\}, s) = 2$. This invariant can show that any system state s' where all philosophers sat at the table is not reachable as $Tks(\{0, 1, 2, 3\}, s') = 3$. Therefore, this invariant captures exactly the behaviour enforced by the semaphore-based mechanism this system implements, and it is strong enough to prove that $\mathcal{S}$ is deadlock free. $\mathcal{S}$ models the solution system for three forks/philosophers but similar arguments and token structures can be made for solutions with $N \geq 4$ forks/philosophers. Note that the binary conservative technique cannot find this structure and invariant. So, this example demonstrates

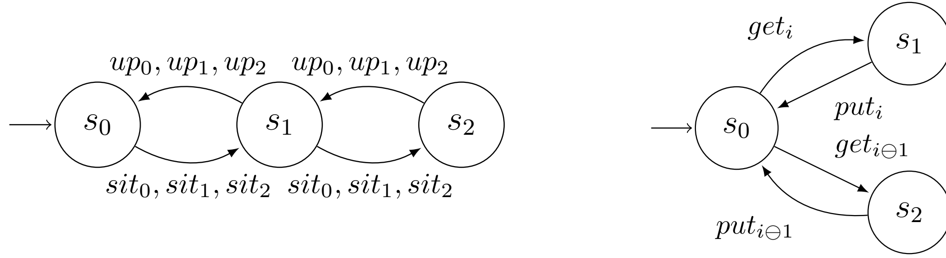

Figure 5.4: LTSs of butler B and of fork F_i, respectively.

that our generalisation to allow components to hold a non-negative number of tokens enhances the capabilities of our framework. ■

It is not difficult to expand this token interpretation to other mechanisms. For instance, a system could employ an (majority-of-votes) election mechanism to elect a leader. We could have a setting where initially each component has a vote (token) to cast. These votes are cast (tokens are passed around) until some component gets the majority of tokens and becomes the leader. This setting clearly describe a token mechanism that would derive the invariant that a reachable state must have as many tokens as there are components in the system. So, for instance, our techniques could prove that a system state where two components claim to be leaders is not reachable, as this violates the token invariant.

Existential token structures can be understood as marking the states where components satisfy a given local property, so they should be used to verify properties that are ensured by the (invariant) fact that at least one component always has this local property (i.e. a token). For instance, in Example 5.2, tokens represent that components are active, and the invariant that a component is always active ensures deadlock freedom. Another interpretation for tokens in an existential structure is that they mark states where a component is not full. This interpretation helps to understand how our existential techniques can capture the non-fillable mechanism discussed in Section 4.3.1, as illustrated by the following example.

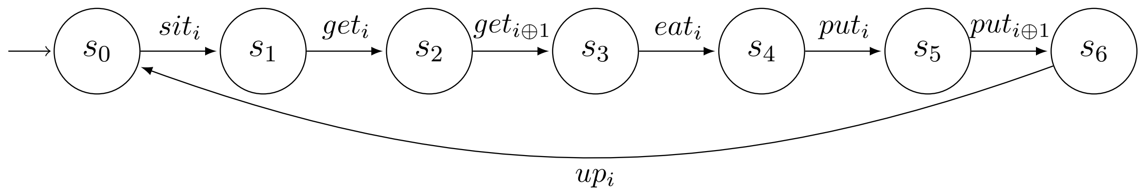

Figure 5.5: LTSs of philosopher P_i.

Example 5.5. This example describes a non-fillable system similar to the ones described in Examples 4.1 and 4.6. Unlike these examples, however, the system we present here does not require components to communicate in a unidirectional ring-like fashion. Instead, a component can pass a token to any other component in the network. A component receives a token from its predecessor component in the ring or from its user, and it either passes the token over to the next component in the ring or outputs the token to its user. Each component can hold up to two tokens at a time and the second token must have been received from its predecessor and not from its user.

This system is described by machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 5.6 and $\mathcal{R}$ the set of rules that requires components to synchronise on shared events except for τ that can be performed independently. Component i can receive a message (i.e. a token) either from component j, via event $tk_{j,i}$, or from its user, via event in_i. If it holds a message, it can pass the message to component j, via event $tk_{i,j}$, or output the message to its user, via out_i. The τ transitions represent an internal (non-deterministic) decision of the component.

For this system, our existential techniques can find $partitions = \{\{0, 1, 2\}\}$ and

$$\begin{aligned} structure = \ & \{(0, s_0), (0, s_1), (0, s_2), (0, s_3), (1, s_0), (1, s_1), \\ & (1, s_2), (1, s_3), (2, s_0), (2, s_1), (2, s_2), (2, s_3)\} \end{aligned}$$

For conciseness, we leave out component states assigned to *false* (i.e. without a token). This structure leads to approximation $reach_{E_\mathcal{B}}(s) \mathrel{\widehat{=}} Tks(\{0, 1, 2\}, s) \geq 1$. Note how tokens mark the states in which a component is *not* full and the invariant captures the non-fillable property: all components cannot be full at the same time. The only blocked state this system admits, (s_6, s_6, s_6), has all components full. So, this invariant can prove it unreachable, as $Tks(\{0, 1, 2\}, (s_6, s_6, s_6)) = 0$, and consequently that $\mathcal{S}$ is deadlock-free. $\mathcal{S}$ is a token network with three components, each of them has a two-slot buffer to store messages. This technique can detect similar structures and invariants that show deadlock freedom for non-fillable systems with $N \geq 3$ components with b-slot buffers where $b \geq 2$. ■

The synchronisation-analysis techniques introduced in the previous chapter can detect some non-fillable, systolic-array-like, and conservative token mechanisms. Their ability to recognise these mechanisms, however, heavily depends on the structure/behaviour of individual components. To some extent, if the system as a whole implements one of these mechanisms but components take part on it in an unorthodox way (i.e. in a way that does not clearly maintain the invariants these mechanism enforce), these techniques are unable to capture them (the same is true for FSDD

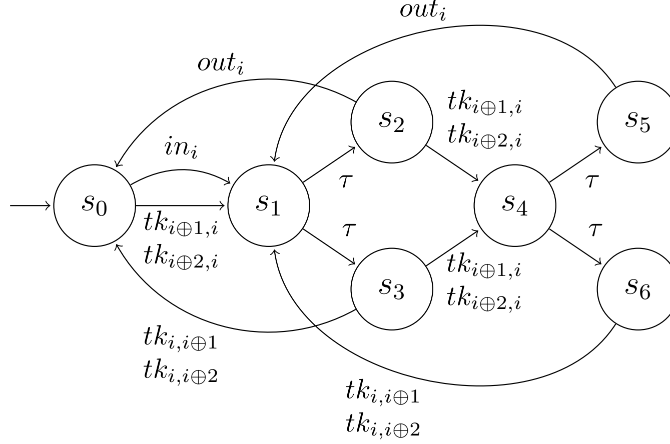

Figure 5.6: LTS of component L_i where $\oplus$ represents addition modulo 3.

and CSDD and the behaviour of pairs of components). We discuss and illustrate this inability in the context of capturing conservative token mechanisms as follows.

The synchronisation-analysis technique giving rise to $reach_D$ (and its abstractions) can compute component-state invariants that capture the balance of token exchanges. The consistency notion this technique relies on can be understood as requiring these balances to add up to zero: token conservation. Its entire ability in recognising such invariants, however, rests on whether they can identify which system rules components use to input and output tokens (so they can accurately capture these balances). This identification means that some rule partitioning puts rules for inputting tokens in one partition and rules for outputting in another. Without this identification, the underlying conservative token structure is missed. We have proposed three different approaches to partition system rules: the trivial partitioning where each rule is in a partition, the data abstraction partitioning where rules with the exact same participants are identified, and a component-specific partitioning based on a component's structure that identifies rules leading to transitions with the same source and target states. There are some systems that implement a token-conservation mechanism for which neither partitioning approaches correctly identifies input and output rules. Intuitively, for these systems, token routes are so unpredictable that these partitionings cannot capture/identify them. The following example illustrates such a system.

Example 5.6. This example presents a token-based scheduler. In this token network, a scheduler component decides which worker component should work next. The scheduler passes a token to this worker and then does some processing. The worker

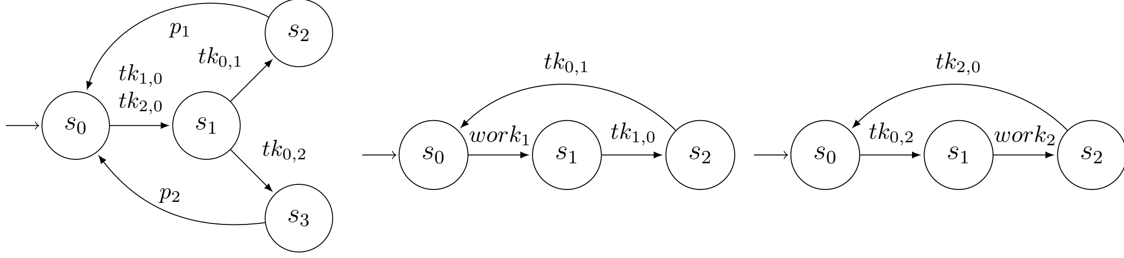

Figure 5.7: LTSs of components L_0, L_1, and L_2, respectively.

component, then, does some work and returns the token to the scheduler. At this point, the scheduler component is ready to activate some worker component again.

This system is implemented by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 5.7 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. An arrow with two labels represents two transitions with the same source and target states but with different labels. Component L_0 is the scheduler and L_1 and L_2 are workers. Worker L_1 has the token (i.e. is activated) initially and event $tk_{i,j}$ represents the passage of a token from L_i to L_j.

None of the synchronisation-analysis techniques proposed in the previous chapter can capture the underlying token mechanism implemented by this system. All of the partitionings they rely on fail to identify the token routes around this network. L_0 uses more than one rule/event to input tokens and the same goes for outputting. Hence, the trivial partitioning cannot capture meaningful token input-output relationships between rules. Moreover, L_0 can exchange tokens with multiple partners and it can do so in an unpredictable way, that is, at state s_1 it chooses between two transitions each of which leads to a particular behaviour and involves a different communication partner. So, neither our data-abstraction partitioning nor our component-specific abstraction can capture meaningful token input-output relationships. Therefore, for this system, all these balance-based techniques create useless approximations.

On the other hand, our binary conservative approach can find the underlying conservative token structure. FINDSTRUCTURE($\mathcal{S}$) can result in $partitions = \{\{0, 1, 2\}\}$ and $structure = \{(0, s_1), (1, s_0), (1, s_1), (2, s_1), (2, s_2)\}$. With this information, we create the invariant $reach_{C_{\mathbb{B}}}(s) \mathrel{\widehat{=}} Tks(\{0, 1, 2\}, s) = 1$. This invariant can show that this system can never be either filled with tokens, as in (s_1, s_1, s_2), or empty, as in (s_0, s_2, s_0). For instance, (s_1, s_1, s_2) is proved unreachable as $Tks(\{0, 1, 2\}, (s_1, s_1, s_2)) = 3$. As these are the two cases in which this system is blocked, this technique can prove that $\mathcal{S}$ is deadlock-free. This system can be understood as a pair of rings, in case two

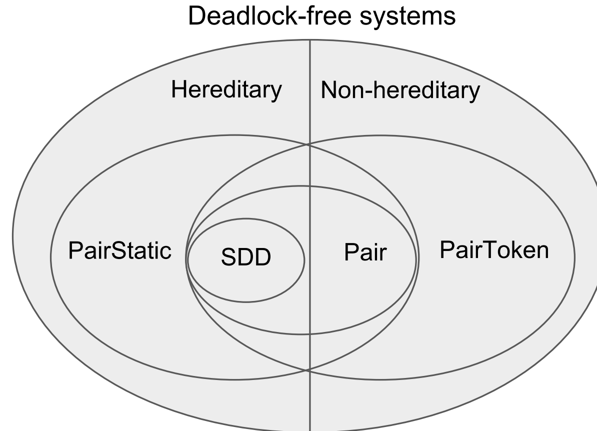

Figure 5.8: The relationship between deadlock-freedom-checking frameworks.

one-process ring in L_1 and L_2, linked by a switch-like process, in this case L_0. This technique should show that rings with N and N' components and $0 < n \leq min(\{N, N'\})$ tokens are deadlock-free. ■

Similar observations can be made for the way in which our synchronisation-analysis techniques capture non-fillable systems; the creation of our existential techniques was motivated by their lack of precision in recognising this mechanism.

Unlike those techniques, the detection techniques that we propose here are oblivious to the structure/behaviour of components and can always find a token mechanism, be it conservative or existential, if the system implements one. They detect token mechanisms based on system transitions rather than some notion of consistency between the component invariants. So, instead of finding token structures as a by-product of a "good" (precise) analysis of components, they directly look for an underlying token structure which respects a conservative/existential invariant. These detection techniques tend to provide a better approximation if compared to our synchronisation-analysis techniques for systems/properties that rely on token structures. Synchronisation analysis, however, more tightly approximates the behaviour of systolic-array-like systems.

In this chapter, we are particularly focused on how these mechanisms and the invariants they maintain can be used to check deadlock freedom. It should not be difficult, however, to extrapolate their application to checking other properties. For instance, token mechanisms and semaphores often ensure mutual-exclusion properties [Pet77, Dij68, AS83]. Figure 5.8 illustrates the relationship between the precision of SDD, Pair, PairStatic and PairToken.

By conjoining these predicates to our Pair candidate definition, we create a more precise characterisation and a corresponding detection problem that is roughly as computationally complex as detecting Pair candidates. Our complexity analysis, however, factors out the detection of token structures by assuming they have already been computed. Detecting/computing token structures should not be that simple but solvers can detect them quite efficiently, as we discuss later.

Theorem 5.17. *The problem of deciding whether a supercombinator machine has a system state satisfying $candidate_{\mathcal{B}}$ or $candidate_{ceg}$ is* NP*-complete.*

Proof. Assuming that a system state s has been fixed, checking $reach_{C_{\mathbb{B}}}(s)$, $reach_{E_{\mathbb{B}}}(s)$, $reach^{ce}_{C}(s)$ or $reach^{ce}_{E_{\mathbb{B}}}(s)$ boils down to summing up some values and carrying out some simple comparisons involving the result of these sums; so each of them can be checked in linear time on the size of the system. So, membership to *NP* follows from this fact and Lemma 3.9, whereas *NP*-hardness follows from Corollary 3.20. □

5.3.2 PairToken-candidate detection via SAT/SMT solving

We built upon our SAT/SMT-checking approach proposed for detecting candidates to create an efficient implementation for PairToken. We implement our framework using SAT formula $PairToken_{\mathbb{B}}$ and SMT formula $PairToken_{ceg}$. While $PairToken_{\mathbb{B}}$ captures system states that satisfy $candidate_{\mathbb{B}}$, $PairToken_{ceg}$ detects states satisfying $candidate_{ceg}$. We use the theory of linear integer arithmetic to encode $PairToken_{ceg}$, and in particular predicate $reach^{ce}_{C}$. So, we encode the search for a 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 where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $ce, ce' \in S$ two counter-example states.

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 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. We reuse the sub-formulas for *State*, *Blocked* and $Reach_2$ defined in Section 3.3.3.

$$PairToken_{\mathbb{B}} \mathrel{\widehat{=}} State \wedge Blocked \wedge Reach_2 \wedge Reach_{C_{\mathbb{B}}} \wedge Reach_{E_{\mathbb{B}}}$$

$$PairToken_{ceg} \mathrel{\widehat{=}} State \wedge Blocked \wedge Reach_2 \wedge Reach^{ce}_{C} \wedge Reach^{ce'}_{E_{\mathbb{B}}}$$

Each of the sub-formulas $Reach_{C_{\mathbb{B}}}$, $Reach_{E_{\mathbb{B}}}$, $Reach_C^{ce'}$ and $Reach_{E_{\mathbb{B}}}^{ce}$ is encoded using two constraints *Structure* and *Cardinality*. The former encodes the predicate's associated token structure and the latter its (sum or existential) cardinality constraint. These constraints use variables tk_i to convey the number of tokens held by component i. Each sub-formula uses a different set of tk_i variables but constraints *Structure* and *Cardinality* of the same sub-formula share their tk_i variables. In the following, when presenting the encoding of a predicate, we use *structure* and *partitions* to denote the associated sets used to construct it.

For $Reach_{C_{\mathbb{B}}}$, $Reach_{E_{\mathbb{B}}}$ and $Reach_{E_{\mathbb{B}}}^{ce'}$, *Structure* is constructed as follows. It relies on boolean variable tk_i to capture whether component i holds a token, according to *structure*.

$$Structure \mathrel{\widehat{=}} \bigwedge_{i\in\{1\ldots n\}\wedge s\in S_i} st_{i,s} \rightarrow \begin{cases} tk_i & \textbf{if } (i,s,true) \in structure \\ \neg tk_i & \textbf{if } (i,s,false) \in structure \end{cases}$$

For $Reach_C^{ce}$, we propose the following *Structure* constraint. It uses natural-number variables tk_i, namely, integer variables restricted by $tk_{i,s} \geq 0$.

$$Structure \mathrel{\widehat{=}} \bigwedge_{i\in\{1\ldots n\}\wedge s\in S_i} st_{i,s} \rightarrow tk_i = structure(i,s)$$

The cardinality constraint for $Reach_{C_{\mathbb{B}}}$, presented next, uses variables tk_i to make sure that, in a satisfying assignment, subsystems in *partitions* have their expected number of tokens. Let sub be a subsystem in *partitions*, $\overline{tk}_{sub}$ the vector of variables tk_i such that $i \in sub$, $\overline{x}_{sub}$ a vector of fresh boolean variable of size $|sub|$, and $N_{sub} = \sum_{i\in sub} structure(i,\hat{s}_i)$ the number of tokens confined in sub. Constraint $Sort(\overline{tk}_{sub}, \overline{x}_{sub})$ makes sure that $\overline{x}_{sub}$ is the result of sorting the values assigned to $\overline{tk}_{sub}$, i.e. true values come first. We use odd-even-merging sorting networks [Bat68] to implement this sorting; they tend to provide a better compromise between the size of the encoding and the efficiency in which these constraints are checked [ES06]. Intuitively, $\overline{tk}_{sub}$ is a unary-unordered representation of the number of tokens being held by components in sub, whereas $\overline{x}_{sub}$ gives its unary-ordered representation. Constraint $Eq(\overline{x}_{sub}, N_{sub})$ ensures that $\overline{x}_{sub}$ is the unary-ordered representation of number N_{sub}.

$$Cardinality \mathrel{\widehat{=}} \bigwedge_{sub\in partitions} Sort(\overline{tk}_{sub}, \overline{x}_{sub}) \wedge Eq(\overline{x}_{sub}, N_{sub})$$

For instance, if in a satisfying assignment we have $\overline{tk}_{sub} = (true, false, true)$ (i.e. 101, a unary-unordered representation of 2), *Sort* makes sure that $\overline{x}_{sub} = (true, true, false)$ (i.e. 110, *the* unary-ordered representation of 2).

The cardinality constraint for $Reach_{E_{\mathbb{B}}}$ and $Reach_{E_{\mathbb{B}}}^{ce'}$ use variables tk_i to ensure that, in a satisfying assignment, subsystems in *partitions* have at least one token. The "at least one token is being held" restriction is a trivial case of a cardinality constraint that can be implemented without need to sorting networks.

$$Cardinality \mathrel{\widehat{=}} \bigwedge_{sub \in partitions} \left(\bigvee_{i \in sub} tk_i \right)$$

The cardinality constraint for $Reach_C^{ce}$ uses linear integer arithmetic to encode the sum invariant enforced by $reach_C^{ce}$ as follows.

$$Cardinality \mathrel{\widehat{=}} \bigwedge_{sub \in partitions} \sum_{i \in sub} tk_i \neq \sum_{i \in sub} structure(i, \hat{s}_i)$$

So, a satisfying assignment to our formulas implies that a candidate exist, whereas unsatisfiability ensures no candidates exist and so the system is deadlock free.

5.3.3 Practical evaluation

We further extend DeadlOx to implement PairToken. It uses solvers Glucose and Z3 to check $PairToken_{\mathbb{B}}$ and $PairToken_{ceg}$, respectively, in an incremental way. It, first, checks the combination of $State$, $Blocked$ and $Reach_2$, and the other approximations are conjoined as follows. For $PairToken_{\mathbb{B}}$, we conjoin first $Reach_{C_{\mathbb{B}}}$ and then $Reach_{E_{\mathbb{B}}}$. For $PairToken_{ceg}$, we add $Reach_C^{ce}$ and then $Reach_{E_{\mathbb{B}}}^{ce'}$. We also use this incremental process to find counter-examples ce and ce'. In checking $PairToken_{ceg}$, we use the first candidate ce in this process to construct $Reach_C^{ce}$ and the second ce' to construct $Reach_{E_{\mathbb{B}}}^{ce'}$, if they exist. We extend FDR4's input language with the annotations `:[PairToken]` and `:[PairToken [smt]]`, which should be added to a deadlock free assertion, to call DeadlOx's $PairToken_{\mathbb{B}}$ and $PairToken_{ceg}$ frameworks, respectively. DeadlOx and the models used in this section are available at [AGRR18].

We analyse some triple-disjoint deadlock-free systems that cannot be tackled by local analysis alone. Hence, Pair and SDD are unable to show deadlock freedom for all examples discussed in this section. We used a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, 8GB of RAM. This experiment compares our frameworks $PairToken_{\mathbb{B}}$ (PT) and $PairToken_{ceg}$ (PTsmt) against our PairStatic's frameworks (PS, PSsmt), the Deadlock Checker's FSDD and CSDD, FDR4's methods (FDR, FDRc, FDRp), and D-Finder 2's methods (DF2pm, DF2fp, DF2l).

Our results again suggest that approximate frameworks are much more scalable than exact ones. Despite being an approximate technique, however, D-Finder 2

performs poorly on almost all examples considered in this section. It seems that the calculations of invariants they carry out is rather complex for these examples. Also, it might be the case that our generation of BIP models does not provide an optimal encoding for D-Finder 2. We point out that the verification time displayed for $PairToken_{\mathbb{B}}$ and $PairToken_{ceg}$ takes into account the detection of token structures, the derivation of invariants and carrying out deadlock-candidate search.

Table 5.1 presents 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) [GS10], a matrix multiplication system (Mat), a ring system implementing a priority-based mutual-exclusion mechanism (RingP), a system using a priority queue to ensure mutual exclusion (PQ), a non-fillable ring (Ring), Milner's scheduler (Sched), and a system that implements a majority-vote mutual-exclusion mechanism (MVote). We do not show the running time of D-Finder 2's techniques as they are much slower than all other approximate ones.

The systems in this table implement a variety of mechanisms to ensure deadlock freedom, and it is quite remarkable that PairToken can capture most of these mechanisms. This means that most of them can be encoded via token structures and invariants. FSDD and CSDD were designed to show non-fillable systems, such as Ring and systolic-array-like systems, such as HexSys, Mat and Sched, respectively, deadlock free. So, all other examples implement mechanisms that are beyond the reach of FSDD and CSDD. $PairToken_{ceg}$ checking seems to be much less scalable than $PairToken_{\mathbb{B}}$ and both PairStatic encodings. The reason seems to be that the detection of structures is not as quick in the SMT setting as it is in the SAT one. Some preliminary study shows that the convergence in the minimisation steps is much slower for $PairToken_{ceg}$ than it is for $PairToken_{\mathbb{B}}$.

PairToken can show deadlock freedom for some counting-based and priority-based systems such as But, RingP and PQ, but it cannot capture the conservation of votes behind the majority-vote mechanism implemented in MVote. To capture the conservation of votes for this example, we would need many token structures but our counter-example-guided conservative technique only detects one. These examples also show the unpredictability of our method. For Mat some instances of Mat and HexSys, it cannot prove deadlock freedom because it finds the "wrong" invariant.

Table 5.2 presents the analysis of some conservative 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 (Ring2Hf), two simplified versions of these two systems (Ring2S and Ring2SHf), a train-track system with one train (Track) and a

		Approximate						Exact		
Example	N	FSDD	CSDD	PS	PSsmt	PT	PTsmt	FDR	FDRc	FDRp
But	50	-	-	-	1.07	-	19.80	*	*	*
	100	-	-	-	4.87	-	*	*	*	*
	150	-	-	-	17.39	-	*	*	*	*
	200	*	*	-	32.52	-	*	*	*	*
DDB	5	-	-	0.31	0.36	-	0.97	0.51	0.16	0.21
	10	*	*	1.27	7.78	-	50.76	*	*	*
	15	*	*	10.53	79.41	-	-	*	*	*
	20	*	*	48.91	*	-	*	*	*	*
HexSys	3	-	0.20	0.16	0.16	-	-	*	0.16	0.36
	5	-	0.28	3.57	0.67	-	-	*	7.33	*
	8	-	0.53	56.12	8.13	-	-	*	*	*
	10	-	0.68	160.63	19.55	-	-	*	*	*
Mat	5	-	0.18	0.36	0.16	-	0.22	*	0.21	0.17
	10	-	0.23	3.87	0.72	-	1.07	*	15.59	0.67
	20	-	0.43	38.84	12.24	-	-	*	*	28.51
	30	-	0.83	269.55	183.45	-	-	*	*	*
RingP	5	-	-	0.22	0.16	0.11	0.31	0.26	+	0.16
	10	-	-	0.41	2.32	0.77	8.63	*	+	*
	15	-	-	2.32	24.21	4.67	203.38	*	+	*
	20	*	*	10.43	158.51	19.40	*	*	+	*
PQ	3	-	-	-	-	-	0.21	0.06	+	0.11
	5	-	-	-	-	-	35.43	0.51	+	0.87
	8	-	-	-	-	-	*	*	+	*
	10	-	-	-	*	-	*	*	+	*
Ring	100	0.18	-	0.17	0.26	0.21	0.46	*	0.72	*
	200	0.28	-	0.26	0.47	0.36	0.92	*	1.57	*
	300	0.38	-	0.42	0.61	0.62	1.32	*	2.72	*
	400	0.43	-	0.51	0.87	0.97	2.02	*	4.22	*
Sched	100	-	0.18	0.11	0.16	0.11	0.42	*	0.37	0.41
	200	-	0.28	0.21	0.26	0.21	3.32	*	0.72	3.82
	300	-	0.38	0.31	0.31	0.31	5.32	*	1.17	15.74
	400	-	0.43	0.47	0.46	0.42	14.99	*	1.77	44.68
MVote	3	-	-	-	-	-	0.11	0.06	0.11	0.11
	5	-	-	-	-	-	-	0.11	0.31	0.16
	7	-	-	-	-	-	-	24.46	*	75.55
	10	-	-	-	-	-	-	*	*	*

Table 5.1: Techniques comparison. 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 that 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.

		Approximate					Exact		
Example	N	PS	PSsmt	PT	PTsmt	DF2	FDR	FDRc	FDRp
MsgGrid	20	-	0.16	0.11	0.26	15.49	*	0.26	*
	40	-	0.21	0.16	0.46	*	*	0.77	*
	60	-	0.26	0.26	0.87	*	*	1.67	*
	80	-	0.36	0.31	1.22	*	*	3.82	*
Ring2	10	0.72	1.47	-	1.52	14.69	0.37	+	0.46
	15	1.92	14.99	-	12.44	*	0.46	+	5.27
	20	10.88	79.67	-	57.98	*	1.07	+	35.87
	25	41.49	*	-	215.52	*	2.87	+	162.71
Ring2Hf	10	0.31	1.42	-	1.72	*	*	+	*
	15	1.92	14.24	-	12.04	*	*	+	*
	20	10.78	83.42	-	79.57	*	*	+	*
	25	41.29	*	-	*	*	*	+	*
Ring2S	50	1.27	2.17	1.07	10.48	217.61	11.60	+	45.03
	100	4.97	22.26	8.13	*	*	*	+	*
	150	19.86	76.16	31.07	*	*	*	+	*
	200	55.62	261.42	83.92	*	*	*	+	*
Ring2SHf	50	1.17	2.62	1.07	8.23	*	*	+	*
	100	4.97	22.31	8.03	115.14	*	*	+	*
	150	20.00	75.06	30.92	*	*	*	+	*
	200	56.02	*	85.35	*	*	*	+	*
Track	100	-	-	0.26	1.12	*	1.32	20.05	2.17
	200	-	-	0.52	6.08	*	9.18	211.58	23.30
	300	-	-	0.82	22.35	*	33.93	*	105.22
	400	-	-	1.17	54.92	*	93.79	*	*
TrackHf	100	-	-	0.26	2.12	*	*	20.05	*
	200	-	-	0.56	7.93	*	*	211.27	*
	300	-	-	0.92	26.81	*	*	*	*
	400	-	-	1.27	52.41	*	*	*	*

Table 5.2: Techniques comparison. 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.

train-track system with $N/2$ trains (TrackHf); trains can be seen as tokens moving around a network of tracks. Neither FSDD nor CSDD can tackle token-based systems so they cannot prove any of these systems deadlock free. Note that Track and TrackHf allow token routes that are too unpredictable for PairStatic to capture. PairToken, however, can capture these routes and derive that trains are conserved. This invariant, in turn, proves these systems deadlock free.

		Approximate		Exact		
Example	N	PSsmt	PTsmt	FDR	FDRc	FDRp
TkFully	10	0.16	0.21	0.11	0.16	0.16
	20	0.52	1.42	0.26	0.31	2.22
	30	1.82	5.92	0.82	0.77	22.03
	40	5.02	19.85	2.82	1.37	117.99
TkFullyHf	10	0.16	0.27	45.85	0.16	238.96
	20	0.57	1.52	*	0.31	*
	30	1.92	7.58	*	0.77	*
	40	5.12	20.00	*	1.37	*
TkGrid	40	0.11	0.27	0.12	+	0.21
	60	0.16	0.21	0.11	+	0.46
	80	0.21	0.31	0.16	+	1.02
	100	0.21	0.36	0.21	+	1.97
TkGridHf	40	0.16	0.21	*	*	*
	60	0.26	0.31	*	*	*
	80	0.32	0.52	*	*	*
	100	0.26	0.47	*	*	*
TkRing	40	0.11	0.21	0.11	+	0.11
	60	0.11	0.26	0.12	+	0.16
	80	0.17	0.56	0.16	+	0.22
	100	0.11	0.77	0.16	+	0.31
TkRingHf	40	0.11	0.17	*	0.46	0.31
	60	0.16	0.21	*	0.87	0.56
	80	0.16	0.26	*	1.72	0.97
	100	0.16	0.32	*	3.17	1.52

Table 5.3: Techniques comparison. 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.

Tables 5.3 and Table 5.4 present the analysis of conservative and existential token networks, respectively. We investigate networks implementing three communication topologies: fully-connected, grid, bidirectional ring. The name of our examples describe the topology used. For examples suffixed by *Hf*, components exchange $N/2$ tokens. Otherwise they exchange only two. Unlike systems in Table 5.2, components in these networks choose a partner to send a token to non-deterministically. These results show that PairToken's precision is neither affected by the different topologies nor by this non-deterministic choice of partner. We only present the results for the approximate techniques that could prove systems deadlock free; the missing ones cannot prove any

of these instances deadlock free.

		Approximate	Exact		
Example	N	PTsmt	FDR	FDRc	FDRp
TkDestFully	5	0.16	0.06	0.11	0.06
	10	0.71	0.11	0.16	0.16
	20	10.83	0.31	0.47	1.52
	30	85.26	0.87	1.87	12.93
TkDestFullyHf	5	0.16	0.06	0.11	0.11
	10	0.87	62.84	0.17	293.30
	20	25.06	*	0.46	*
	30	105.91	*	1.87	*
TkDestGrid	20	0.16	0.06	+	0.11
	30	0.16	0.11	+	0.11
	40	0.21	0.11	+	0.21
	50	0.26	0.11	+	0.31
TkDestGridHf	20	0.21	*	17.35	*
	30	0.21	*	*	*
	40	0.26	*	*	*
	50	0.31	*	*	*
TkDestRing	20	0.11	0.11	+	0.12
	30	0.11	0.11	+	0.11
	40	0.11	0.11	+	0.11
	50	0.16	0.11	+	0.16
TkDestRingHf	20	0.11	*	0.21	21.81
	30	0.16	*	0.31	*
	40	0.17	*	0.46	*
	50	0.16	*	0.66	*

Table 5.4: Techniques comparison. 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.

In Table 5.5, we analyse two networks (Cons and Cons2) that generalise the conservative token ring in TkRing2 and two networks (NonCons and NonCons10) that generalise the non-filable ring in Ring. They turn these examples into fully connected networks. In Cons, components exchange one token, whereas in Cons2, they exchange $N/2$ tokens. In NonCons, components have a one-slot buffer to store messages, whereas in NonCons10, they have a 10-slot buffer. These examples show that PairToken is able to recognise the conservative invariant of Cons and Cons2 and the existential one of NonCons and NonCons10. Both these invariants are enough

		Approximate	Exact		
Example	N	PTsmt	FDR	FDRc	FDRp
Cons	20	4.02	0.92	+	33.68
	30	25.61	5.72	+	*
	40	115.59	36.49	+	*
	50	*	134.45	+	*
Cons2	20	4.02	*	+	*
	30	25.81	*	+	*
	40	116.74	*	+	*
	50	*	*	+	*
NonCons	5	0.11	0.16	+	0.17
	10	0.26	*	+	*
	20	1.02	*	+	*
	30	2.97	*	+	*
NonCons10	5	1.72	88.14	+	235.47
	10	13.44	*	+	*
	15	45.74	*	+	*
	20	199.03	*	+	*

Table 5.5: Techniques comparison. 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.

to prove these systems deadlock free. PairStatic, however, cannot recognise these invariants and it is so unable to prove these systems deadlock free.

These results show that PairToken can tackle a relevant class of distributed and concurrent systems. Generally, it seems that PairToken is less scalable than PairStatic but it can handle more examples. PairToken can, in particular, handle systems implementing token routes that are too unpredictable for PairStatic to handle. Proving deadlock freedom for these examples is far from trivial; they rely on rather intricate mechanisms that are recognised by PairToken. It is quite remarkable how many non-trivial interaction mechanisms, which are not obviously token-based, can be captured by our structures. We did not anticipate, for instance, the token structures discovered for the Mat example. The sort of systolic-array invariant needed to prove deadlock freedom for this systems is not evidently represented by a token structure.

These results also seem to suggest that our SMT-based approach is not as scalable as other approximate techniques. The detection of token structures is not as quick as it is in the SAT setting. The minimisation steps seem to converge far slower in the SMT case than they do in the SAT case, hindering the verification of our

$PairToken_{ceg}$ encoding. Moreover, in many cases (like for our MVote, Mat and HexSys examples), more intricate invariants can be represented by a combination of many token structures. In this sense, our counter-example-guided detection could be adapted to implement a proper CEGAR-like loop, where new counter-examples would be used to find new token structures until either no new structures could be detected, and we report back the final counter-example, or the property is proved. As many CEGAR-like frameworks, this adaptation could only succeed with a strategy that makes this loop converge quickly. We could not, however, devise such a strategy. We point out compression techniques can be applied very successfully to many examples. Nevertheless, their effective use requires a careful and skilful application of those, whereas our method is fully automatic.

5.4 Conclusion

This chapter investigates how token structures and invariants can be used to the effective verification of concurrent systems.

Our techniques rely on a SAT/SMT specification of a token structure to detect whether a system implements one. From this structure, it derives an invariant that serves as a reachability approximation. We have identified two types of token structures: the first one makes sure that tokens are conserved, and the second one ensures at least one token is present in the system at all times. They can detect token structures sometimes too subtle to be obviously recognisable as such. In fact, as our experiments show, many interaction mechanisms that are not evidently token based can be captured by token structures. Moreover, while we have interpreted these structures as token mechanisms, there might be other views to them. For instance, as we discussed in the application of our existential technique to Example 5.5, tokens can be seen as the component property "component is not full".

We detect virtual tokens that are indistinguishable abstract elements of the mechanism used by components to cooperate. So, we have no particular way of distinguishing different types/classes of tokens. That said, we should be able to detect the relationships between different classes of tokens using our techniques without noticing/capturing that tokens are of different classes. The only requirement for capturing these relationships is that they must be framed into the conservative or existential policies our token structures require. We could, for instance, adapt our techniques to detect many token structures for the same part of system. Each of these different token structures capture different relationships between tokens which could,

in principle, relate tokens of different classes. There is no way, however, to guarantee that we find structures relating different classes of tokens because we do not have a notion of token classes/types to begin with.

We combine these token invariants/approximations with 2-reachability presented in Chapter 3 to create the PairToken framework. These new approximations allow this framework to capture global system invariants. Hence, it can prove deadlock free systems that are beyond the capabilities of Pair. We present some experimental evidence that suggests PairToken is able to prove deadlock freedom for an interesting class of distributed and concurrent system, different from the class tackled by current approximate techniques, and it does so in a rather scalable way. Our experiment also demonstrates that, for the systems analysed, the SAT and SMT calculations used to detect token structures can be carried out rather efficiently, albeit SAT detection is much quicker. Also, PairToken can derive invariants and carry out deadlock-freedom proofs that are far from trivial, attesting its usefulness.

Our reachability approximations were inspired by PairStatic's inability to capture token invariants for systems implementing token routes that are unpredictable. Many concurrent systems use a token mechanism to avoid undesired states so capturing these structures is often vital in proving they behave correctly. Most approximate techniques, however, cannot tackle such systems. CSDD is unable to capture conservative token structures altogether and FSDD can only handle non-fillable rings that are designed in a restricted way. The technique behind D-Finder 2, notably, tries to find invariants similar to ours. It relies on a set of implications that are similar to the *Policy* constraint we propose for our binary existential technique. However, this technique tries to capture far too many token structures; it might construct an invariant that is exponentially large on the size of the input system, rendering the verification of deadlocks infeasible.

Finally, we point out that the token structures that we capture in this work can be interpreted as invariants of a Petri net induced by the concurrent system under analysis. This Petri net has a place (i, s) for each state s of component i, a transition for each individual and pairwise transition induced by the projections in Definition 5.1, and its initial marking places a token at each component state (place) in the system's starting state. A pairwise transition of components i and j from states s_i,s_j to s_i', s_j' (respectively) gives rise to a transition t and arcs from (i, s_i) and (j, s_j) to t and from t to (i, s_i') and (i, s_i'). Similarly, an individual transition of component i from state s_i to s_i' creates a transition t' and arcs from (i, s_i) to t' and from t' to (i, s_i'). This net mimics the behaviour of the system: markings denote system states (the combination

of places with a token forms a system state) and net transitions directly capture system transitions. Our conservative token structures are *place invariants* for this Petri net, whereas our existential structures are *traps* [Mur89, Pet77]. Place invariants are meant to generally capture some linear combinations of places' markings that are invariant. For instance, for a system with places p_1, p_2, p_3, the place invariant $(1, 3, 0)$ indicates transitions must respect the fact that $M(p_1) + 3M(p_2) = M'(p_1) + 3M'(p_2)$, where M (M') is the marking before (after) the transition. So, it has a rather distinct purpose if compared to our token structure. Nevertheless, we can see how the notion of place invariants and conservative token structures and invariants (almost) coincide thanks to the particular way our induced net's markings represent system states. A similar observation can be made about traps, these capture sets of places where the presence of a token is an invariant. Our detection problem, however, also takes into account the reachability-analysis problem we are tackling. For instance, a token structure where a single component always holds a token is a valid place invariant (and trap) of this induced net and yet it leads to a pointless reachability approximation. It captures an invariant that is trivially ensured by the concurrent system's structure and, hence, does not help with the sort of reachability analysis we perform. These new characterisations for our structures could lead to other methods to calculate them; we are yet to explore this connection.

Chapter 6

Approximate verification of static properties

6.1 Introduction

The previous chapters demonstrate how replacing exact reachability by our approximations can speed up the verification of deadlock freedom. A natural question that arises is which other properties can be *effectively* tackled by this approach. This chapter investigates how our approximations can be used to check *static properties*. They specify that "bad states cannot be reachable" where states are deemed *bad* based on their *static* behaviour, namely, some combination of events the system can/cannot perform when in these states. So, verifying these properties naturally fits into the sort of reachability analysis approach we propose.

As bad states are recognised based on the analysis of their available interactions, their detection can be simply encoded into the kind of SAT/SMT constraints we use. By solving the constraints proposed so far, we test whether possible system states meet our encoding/specification of a blocked state. To capture static properties instead of deadlock freedom, we simply need to specify which events are available at different system states and for which combinations of events states are considered bad. This simplicity/compactness in encoding violations makes static properties a natural target for our SAT/SMT-based verification frameworks.

Static properties are also good targets for our sort of framework because they do not interfere with our approximations. Verification frameworks tend to analyse more complex behaviours such as traces of events or failures of a system. In these cases, properties are often specified via an automaton, and property checking boils down to examining the combination of system and property automaton. For such complex properties, our approximations would need to estimate reachability for this

combination. Given their state-based nature, however, static properties can be tested by examining the states of the system alone. So, our approximations need only to estimate reachability for the system itself.

In this chapter, we introduce a notation to describe static properties, and their semantics. Roughly speaking, a static property specifies a combination of offered and refused events that defines whether a system state is bad. We propose a global and a local interpretation for such a specification. For the global one, a system satisfies a static property if it cannot reach a bad state. For the local one, a system satisfies a static property if its subsystems cannot reach a bad state. Deadlock freedom and *local-deadlock freedom*, namely, the property that no subsystem can become irretrievably blocked, are natural examples of these two types of properties. Local-deadlock freedom, in particular, is a commonly desirable property that is quite hard to specify and efficiently check using conventional techniques. Mutual exclusion and safe invocation, namely, ensuring components only invoke services that are properly initialised, are other relevant examples of static properties.

This chapter also introduces *StaticProperty*, an approximate framework to check static properties that generalises our previous frameworks for deadlock-freedom checking. Instead of looking for blocked states, it relies on a constraint that encodes whether a state is bad according to some input static property. Furthermore, it combines 2-reachability with the reachability approximations in both Chapter 4 and Chapter 5 to enhance precision. Hence, it can leverage any interaction mechanisms that can be captured by our approximations to prove static properties. Nevertheless, this framework is still incomplete: when a violation is found, it produces an inconclusive result as it does not know whether this bad state is reachable or not. If no violation is found, however, the system satisfies the static property. We believe this incompleteness is a small price to pay to achieve better scalability. We implement our framework in a tool called *ApprOx*. The efficiency and precision of this tool is assessed by practical experiments where we check some concurrent systems using our implementation. To the best of our knowledge, in the context of concurrent systems, our framework is the first approximate approach to directly tackle such a general class of properties.

This chapter's outline is as follows. In Section 6.2, we introduce a notation to capture static properties and its semantics, whereas in Section 6.3, we propose and investigate an approximate framework to check static properties using SAT and SMT solving. In Section 6.4, we present our concluding remarks.

6.2 Capturing static properties

In this section, we propose a framework to specify *static properties* of a concurrent system. A static property takes the form: "the system never reaches a *bad* state". In our framework, bad states are described based on the system's immediate behaviour. Hence the term static; it does not attempt to address things like traces.

6.2.1 Static properties

A *static property* is described by a pair $(\mathcal{V}, \sim)$ where $\mathcal{V}$ is a *violation formula* and $\sim$ is a *satisfiability relation*. A *violation* for this property is a system state s such that $s \sim \mathcal{V}$. Roughly speaking, $\mathcal{V}$ describes some bad behaviour and $\sim$ is a relation that captures whether a state behaves badly. A violation formula describes some immediate behaviour of a system by a propositional formula where atomic propositions correspond to system events.

Definition 6.1. Let Σ be the alphabet of the system under analysis, which can include τ or $\checkmark$, and $ev \in \Sigma$. A violation formula $\mathcal{V}$ is inductively constructed as follows.

$$\mathcal{V} \mathrel{\widehat{=}} \textbf{Event } ev \mid \textbf{Not } \mathcal{V} \mid \textbf{Or } \mathbb{P}(\mathcal{V}) \mid \textbf{And } \mathbb{P}(\mathcal{V})$$

The following formula is an example of a violation formula:

Violation formula 1. $\mathcal{V}_1 \mathrel{\widehat{=}} \textbf{\textit{And}} \ \{\textbf{\textit{Not Event}}\ ev \mid ev \in \Sigma\}$

We propose two satisfiability relations that give rise to two different types of violations. The first relation is based on the overall system's behaviour. We call violations of this type *global violations*. For this relation, the atomic proposition "**Event** ev" holds for states in which the system can perform event ev.

Definition 6.2. Let $\mathcal{V}$ be a violation formula, $\mathcal{S}$ a supercombinator machine with $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $ev \in \Sigma$. A state s is a global violation for formula $\mathcal{V}$ iff $s \models \mathcal{V}$ holds. Let $\mathcal{V}'$ be a violation formula and $\mathcal{VS}$ be a set of violation formulas, $s \models \mathcal{V}$ can be calculated using the following clauses:

$s \models \textbf{Event}\ \text{ev}$	iff	$s \xrightarrow{ev}$	$s \models \textbf{Or}\ \mathcal{VS}$	iff	$\bigvee_{\mathcal{V}' \in \mathcal{VS}} s \models \mathcal{V}'$
$s \models \textbf{Not}\ \mathcal{V}'$	iff	$s \not\models \mathcal{V}'$	$s \models \textbf{And}\ \mathcal{VS}$	iff	$\bigwedge_{\mathcal{V}' \in \mathcal{VS}} s \models \mathcal{V}'$

This relation and Violation formula 1 characterise the blocked states of a system, i.e. $\{s \in S \mid s \models \mathcal{V}_1\}$ for a system with states S.

The second relation is interpreted based on the behaviour of subsystems. We call violations of this type *local violations*. We analyse the behaviour of a subsystem through the following projection. Given a subsystem, it restricts the behaviour of the system to the transitions involving components in this subsystem.

Definition 6.3. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $ss \subseteq \{1 \ldots n\}$ such that $ss \neq \emptyset$ a subsystem, and e_i be the i-th element of tuple e. The projection of $\mathcal{S}$ on ss is given by $\mathcal{S}_{ss}$ defined as:

$$(\langle L_1, \ldots, L_n \rangle, \{(e_{ss}, a) \mid (e, a) \in \mathcal{R} \land \exists\, i \in ss \bullet e_i \neq -\})$$

where e_{ss} is the tuple e' such that $e'_i = e_i$ if $i \in ss$, and $e'_i = -$ otherwise.

We use $s \xrightarrow{ev}_{ss}$ to denote the predicate $s \xrightarrow{ev}$ with respect to the LTS induced by $\mathcal{S}_{ss}$. For this local relation, a violation occurs when one subsystem engages in some bad behaviour. Hence, a violation formula is evaluated with respect to every subsystem ss of the system under analysis. The atomic proposition "**Event** ev", in particular, holds whenever a subsystem ss can perform ev.

Definition 6.4. Let $\mathcal{V}$ be a violation formula, $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ a supercombinator machine, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $ev \in \Sigma$. A state s satisfies the formula $\mathcal{V}$ under the local interpretation iff $s \mid\models \mathcal{V}$ holds, where $s \mid\models \mathcal{V}$ is a shorthand for $\exists\, ss \subseteq \{1 \ldots n\} \mid ss \neq \emptyset \bullet s \mid\models_{ss} \mathcal{V}$. Let $\mathcal{V}'$ be a violation formula and $\mathcal{VS}$ a set of violation formulas, $s \mid\models_{ss} \mathcal{V}$ can be calculated as follows:

$s \mid\models_{ss}$ **Event** ev	iff	$s \xrightarrow{ev}_{ss}$	$s \mid\models_{ss}$ **Or** $\mathcal{VS}$	iff	$\bigvee_{\mathcal{V}' \in \mathcal{VS}} s \mid\models_{ss} \mathcal{V}'$
$s \mid\models_{ss}$ **Not** $\mathcal{V}'$	iff	$s \mid\not\models_{ss} \mathcal{V}'$	$s \mid\models_{ss}$ **And** $\mathcal{VS}$	iff	$\bigwedge_{\mathcal{V}' \in \mathcal{VS}} s \mid\models_{ss} \mathcal{V}'$

The second relation and Violation formula 1 characterise locally blocked states, namely, $\{s \in S \mid s \mid\models \mathcal{V}_1\}$ are the states for which a subsystem is blocked.

We make a few remarks about these two satisfiability relations. Firstly, one should note that $\models$ is equivalent to $\mid\models_{ss}$ when ss is the set containing all the component indices in the system under analysis. Secondly, we point out that for a given state s and a given violation $\mathcal{V}$, from our first remark and our definition of $\mid\models$, it follows that if $s \models \mathcal{V}$ then $s \mid\models \mathcal{V}$. So, the second relation gives rise to a much weaker/looser sort of violation compared to the first one; it only takes one violating subsystem to establish a violation. Thirdly, we point out that the task of analysing local violations should be more demanding as it involves the analysis of all subsystems of a system.

6.2.2 Checking static properties

A system without reachable violations satisfies the corresponding static property.

Definition 6.5. Let $\mathcal{V}$ be a violation formula, $\sim \in \{\models, |\!\models\}$ a satisfiability relation, $\mathcal{S}$ a supercombinator machine and S the set of states for its induced LTS. The system $\mathcal{S}$ satisfies the static property $(\mathcal{V}, \sim)$ iff $\neg \exists s : S \bullet \mathit{reachable_violation}(s)$ where $\mathit{reachable_violation}(s) \mathrel{\widehat{=}} \mathit{reachable}(s) \wedge s \sim \mathcal{V}$.

For instance, $(\mathcal{V}_1, \models)$ captures deadlock freedom, since reachable violations are deadlocks. $(\mathcal{V}_1, |\!\models)$, on the other hand, captures local-deadlock freedom, as reachable violations represent local deadlocks.

Local static properties, i.e. static properties employing the local relation, should provide an interesting tool to show that a system respects a given property thanks to the good behaviour of its subsystems. Ensuring that all subsystems respect a given property might be more desirable than showing the property for the global system, and this fact is often neglected. For instance, let us examine the relationship between deadlock and local-deadlock freedom. Usually, deadlock freedom is checked as a first step to show that a system behaves as expected. A system, however, might be deadlock free, not because it is well behaved, but because an individual component is always making the system progress, even though the rest of the system is blocked. So, in general, checking local-deadlock freedom, which ensures that no subsystem gets to a state in which it is forever stuck, seems like a better initial step in showing that a system behaves correctly.

We point out that a local static property is stronger than its global counterpart. The local satisfiability relation weakly defines that a violation occurs when one subsystem behaves badly but when plugged into a static property it requires all subsystems to be violation free. Note this generalisation to local properties does not scale well in most verification frameworks. For most frameworks, checking this property would entail carrying out exponentially many checks; one for each (connected) subsystem.

6.2.3 Capturing some other static properties

In this section, we move away from properties involving blocked states and show how to capture mutual-exclusion properties and a sort of safe-invocation property.

It is often the case that concurrent systems implement a mutual-exclusion mechanism to ensure that the system behaves properly. We capture a *general mutual-exclusion property* by the sets CS_i of component states, defining the *critical section* of

component i. So, this property states that no two components can be simultaneously in their critical sections.

To capture this property, components' critical sections must be identified. The special (fresh) event cs_i annotates each state of component i that belongs to its critical section. This annotation is made with a self-loop transition using cs_i. Furthermore, we add some system rules that allow the system to perform this new special event so they can be used in static properties. Note the use of our square-bracketed notation for event tuples.

Definition 6.6. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $CS_i \subseteq S_i$ the critical section of component i, and $cs_i \notin \Sigma_i$. We use the modified system $\mathcal{S}_{MX} \mathrel{\widehat{=}} (\langle L'_1, \ldots, L'_n \rangle, \mathcal{R} \cup \mathcal{R}')$ to verify general mutual-exclusion properties.

- $L'_i = (S_i, \Sigma_i \cup \{cs_i\}, \Delta_i \cup \{(s, cs_i, s) \mid s \in CS_i\}, \hat{s}_i)$
- $\mathcal{R}' = \{([i, cs_i], cs_i) \mid i \in \{1 \ldots n\}\}$

A violation occurs when any two components i and j are simultaneously in their critical sections. In our modified system, this corresponds to a state in which the events cs_i and cs_j can be performed, i.e. a state s satisfying $s \models \mathcal{V}_2$.

Violation formula 2. *For components* $\{1 \ldots n\}$*:*

$$\mathcal{V}_2 \mathrel{\widehat{=}} \textbf{\textit{Or}}\ \{\textbf{\textit{And}}\ \{\textbf{\textit{Event}}\ cs_i, \textbf{\textit{Event}}\ cs_j\} \mid i, j \in \{1 \ldots n\} \wedge i \neq j\}$$

So, a system $\mathcal{S}$ satisfies the general mutual-exclusion property with respects to the critical sections CS_i *iff* the modified system $\mathcal{S}_{MX}$ satisfies $(\mathcal{V}_2, \models)$.

Many systems, however, implement a more fine-grained type of mutual exclusion. For instance, concurrent access to read from a storage space is often not harmful but concurrent access to read and write or to write is. We propose a *read-write mutual-exclusion property* that is specified by the pairs of sets R_i and W_i, which represent the reading and writing sections of component i. We employ the same annotation method as before and use self-loops with events r_i and w_i to annotate the component states in the reading and writing sections, respectively, of component i.

Definition 6.7. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $R_i, W_i \subseteq S_i$ the reading and writing sections of component i, respectively, and $r_i, w_i \notin \Sigma_i$ events. The modified read-write system is given by $\mathcal{S}_{RW} \mathrel{\widehat{=}} (\langle L'_1, \ldots, L'_n \rangle, \mathcal{R} \cup \mathcal{R}')$ where:

- $L'_i = (S_i, \Sigma_i \cup \{r_i, w_i\}, \Delta_i \cup \{(s, r_i, s) \mid s \in R_i\} \cup \{(s, w_i, s) \mid s \in W_i\}, \hat{s}_i)$
- $\mathcal{R}' = \{([i, r_i], r_i), ([i, w_i], w_i) \mid i \in \{1 \dots n\}\}$

A violation to the read-write mutual-exclusion property occurs when a component i is in its writing section and another components j is either in its writing or reading section. Hence, such a violating state s satisfies $s \models \mathcal{V}_3$.

Violation formula 3. *For components* $\{1 \dots n\}$*:*

$$\mathcal{V}_3 \mathrel{\widehat{=}} \textbf{\textit{Or}}\ \{\textbf{\textit{And}}\ \{\textbf{\textit{Event}}\ w_i, \textbf{\textit{Or}}\{\textbf{\textit{Event}}\ r_j, \textbf{\textit{Event}}\ w_j\}\} \mid i, j \in \{1 \dots n\} \land i \neq j\}$$

So, a system $\mathcal{S}$ satisfies the read-write mutual-exclusion property for sections R_i and W_i *iff* the modified system $\mathcal{S}_{RW}$ satisfies the static property $(\mathcal{V}_3, \models)$.

In addition to these two mutual-exclusion properties, we propose a way to ensure components only invoke services that are properly initialised. We describe a *safe-invocation property* by sets I_i. The set I_i contains pairs (ev, U): the event ev represents a service offered by component i and U the states of component i where this service has not been initialised yet. We identify component states in the sets U by annotating them with self-loops using events $si_{i,ev}$.

Definition 6.8. Let $\mathcal{S} = (\langle L_1, \dots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $I_i \subseteq \Sigma_i \times \mathbb{P}(S_i)$, where if $(ev, U), (ev, U') \in I_i$ then $U = U'$, the initialisation requirements for component i, and $si_{i,ev} \notin \Sigma_i$ events. We use the modified system $\mathcal{S}_{SI} \mathrel{\widehat{=}} (\langle L'_1, \dots, L'_n \rangle, \mathcal{R} \cup \mathcal{R}')$ to capture a safe-invocation property.

- $L'_i = (S_i, \Sigma_i \cup \{si_{i,ev}\}, \Delta_i \cup \bigcup_{(ev,U) \in I_i} \{(s, si_{i,ev}, s) \mid s \in U\}, \hat{s}_i)$
- $\mathcal{R}' = \{([i, si_{i,ev}], si_{i,ev}) \mid i \in \{1 \dots n\} \land (ev, U) \in I_i\}$

A violation to the safe-initialisation property occurs when a service ev is invoked but the corresponding provider component is uninitialised, namely, in a component state in U. Such a violating state s satisfies $s \models \mathcal{V}_4$.

Violation formula 4. *For components* $\{1 \dots n\}$ *and sets* I_i*:*

$$\mathcal{V}_4 \mathrel{\widehat{=}} \textbf{\textit{Or}}\ \{\textbf{\textit{And}}\ \{\textbf{\textit{Event}}\ ev, \textbf{\textit{Event}}\ si_{i,ev}\}\} \mid i \in \{1 \dots n\} \land (ev, U) \in I_i\}$$

A system $\mathcal{S}$ satisfies the safe-invocation property for sets I_i if and only if $\mathcal{S}_{SI}$ satisfies the static property $(\mathcal{V}_4, \models)$.

Note that to capture these properties we systematically identify some special states of components and specify a violation based on some combination of these special states. This systematic process hints to the generality of static properties. Any property for which bad combinations of component states can be captured in this way can be expressed as a static property. The design of our notation was driven by the specific properties presented here and the event-based formalism we use to describe systems. Nevertheless, we do not anticipate any difficulties in generalising this language to accommodate, for instance, special sets of marked states (which could replace our annotated states) or quantification in terms of specific subsystems. Note our annotation of states using self-loops with special events is already an implicit representation of a set of marked states, and handling quantification for specific subsystems should be simpler than handling quantification over all subsystems.

6.3 Approximate verification of static properties

One way to check static properties is to explicitly look for a violation in the reachable states of the system. Explicit state exploration, however, tends to be very inefficient due to the state-space explosion problem. In this section, we introduce a verification framework, which we call *StaticProperty*, that builds on our combination of reachability approximations and SAT/SMT checking.

StaticProperty combines our reachability approximations to check static properties. PairStatic and PairToken (introduced in the past two chapters) combine pairwise analysis with synchronisation analysis and with token invariants, respectively. These two frameworks were used to contrast synchronisation analysis and token-invariant detection as techniques to capture global system invariants. Here, instead of contrasting techniques, we want to leverage all our techniques to approximate reachability as tightly as we can. Given this combination of techniques, StaticProperty can tackle systems implementing a combination of the mechanisms captured by each individual technique. We propose two versions for this framework, $StaticProperty_{SAT}$ and $StaticProperty_{SMT}$, each of which rely on a different combination of approximations. While the former conjoins/groups the approximations that were encoded as SAT constraints, the latter combines approximations that were encoded as SMT constraints.

Definition 6.9. Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For a state $s \in S$ and counter-examples $ce, ce' \in S$, we define:

$$
\begin{aligned}
reach_{SAT}(s) \mathrel{\widehat{=}}\ & reach_2(s) \land reach_R(s) \land reach_R^{DA}(s) \land reach_S(s) \\
& \land reach_S^{DA}(s) \land reach_{C_{\mathbb{B}}}(s) \land reach_{E_{\mathbb{B}}}(s)
\end{aligned}
$$

$$
\begin{aligned}
reach_{SMT}(s) \mathrel{\widehat{=}}\ & reach_2(s) \land reach_D(s) \land reach_D^{DA}(s) \land reach_D^{CA}(s) \\
& \land reach_S(s) \land reach_S^{DA}(s) \land reach_C^{ce}(s) \land reach_{E_{\mathbb{B}}}^{ce'}(s)
\end{aligned}
$$

It should be clear that these combinations (i.e conjunctions) of reachability approximations are themselves over-approximations.

Theorem 6.10. *Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. For any state $s \in S$, we have that both $reachable(s) \Rightarrow reach_{SAT}(s)$ and $reachable(s) \Rightarrow reach_{SMT}(s)$ hold.*

Instead of looking through the reachable states of the system, our framework looks for a *candidate violation*, namely, a violating state that passes one of our reachability definitions. Each version relies on a different candidate-violation definition, employing its associated approximation.

Definition 6.11. Let $(S, \Sigma, \Delta, \hat{s})$ be the induced LTS under analysis, and $(\mathcal{V}, \sim)$ a static property. For $s \in S$ and x either SAT or SMT, we have that:

$$candidate_x(s) \mathrel{\widehat{=}} s \sim \mathcal{V} \land reach_x(s)$$

This framework is imprecise as a candidate might pass our reachability test and yet be unreachable. Nevertheless, by conjoining our approximations, we tighten the state space analysed; it only takes one failed test to consider a state unreachable. Furthermore, the use of over-approximations (see Theorem 6.10) makes our framework sound; if no candidates are found, the corresponding static property must hold.

Theorem 6.12. *Given a static property, if a system is candidate-violation free it must also be free of reachable violations.*

6.3.1 Precision and complexity of StaticProperty

Our framework inherit the strengths and weaknesses of our approximations. It proves deadlock freedom for any system employing a mechanism to avoid bad states that can be captured by one (or a combination) of our approximations. For instance, our pairwise local analysis can capture some resource-allocation and client-server mechanisms while our global-analysis techniques can capture some systolic-array, priority-based, token-based and counting mechanisms. Nevertheless, it cannot handle systems that are beyond the reach of all our approximations. Moreover, the use of

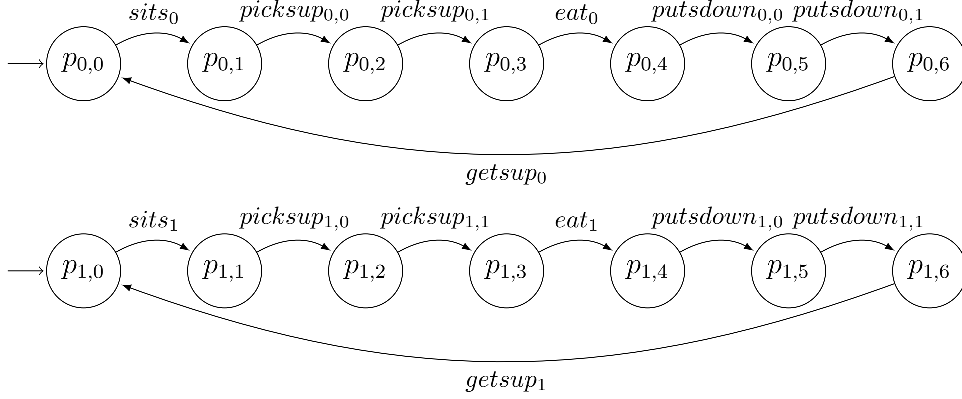

Figure 6.1: LTSs of Philosophers 0 and 1, respectively.

our token-structure-detection techniques also makes this strategy unpredictable in the sense that we might not be able to anticipate which invariant it will be captured.

Next, we present three examples that illustrate the use of our approximations in checking some static properties. The first example demonstrates the use of our pairwise local analysis to verify a system implementing a resource-allocation mechanism.

Example 6.1. This example presents the well-known asymmetric solution to the dining philosophers problem. In this system, $N = 2$ philosophers compete to acquire a pair of forks in order to eat. Each philosopher has to sit on a round table, acquire the fork positioned on its left-hand side and another on its right-hand side, so they can eat. All philosophers acquire first the left-hand-side fork and then the right-hand-side one, except for one of them which acquires the forks in the opposite order.

Let $\mathcal{S} = (\langle L_0, L_1, L_2, L_3\rangle, \mathcal{R})$ be the supercombinator machine such that L_0, L_1 are Philosophers 0 and 1 as described in Figure 6.1, and L_2 and L_3 are Forks 0 and 1 as per Figure 6.2 and $\mathcal{R}$ requires components to synchronise on shared events. Events $sits_i$,$getsup_i$, and eat_i depict the activities of sitting, getting up and eating by Philosopher i, respectively. Events $picksup_{i,i}$ and $picksup_{i,i\oplus 1}$ ($putsdown_{i,i}$ and

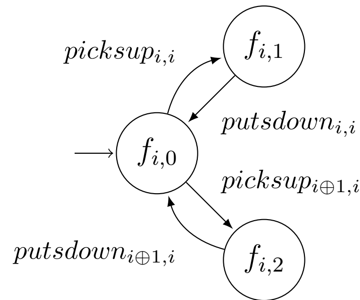

Figure 6.2: LTS of Fork i.

$putsdown_{i,i\oplus 1}$) are used by Philosopher i to acquire (release) the forks on the left-hand side and right-hand side, respectively. $\oplus$ represents addition modulo 2.

We ensure that (adjacent) Philosophers 0 and 1 cannot be eating at the same time by checking the following static property: $(\textbf{And}\,\{\textbf{Event}\, eat_0, \textbf{Event}\, eat_1\}, \models)$. Our framework can show that this system satisfies this static property. The sort of pairwise analysis implemented by $reach_2$ (proposed in Section 3.2.1) can show that any system state involving component states $p_{0,3}$ and $p_{1,3}$ must be unreachable. This version has only two philosophers and two forks but our pairwise analysis could show this property for any version with N philosophers and N forks. Also, it could show this property no matter which two adjacent philosophers are picked. ■

The second example illustrates how synchronisation analysis can be used to verify a non-fillable system.

Example 6.2 (From [Ros10])**.** This example describes a non-fillable system where components exchange messages unidirectionally around the ring they form. A component receives messages from its predecessor component in the ring or from its user, and it either passes the message over to the next component in the ring or outputs the message to its user. Each component can hold up to two messages at a time and the second message it holds must have been received from its predecessor and not from its user. This system is captured by supercombinator machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 6.3 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. A component receives messages from its predecessor via event $ring_i$, and from its user via event in_i. It can pass a message along to the next component in the ring via event $ring_{i\oplus 1}$, and output a message to its user via out_i. The τ transitions represent a non-deterministic choice of the component. The event

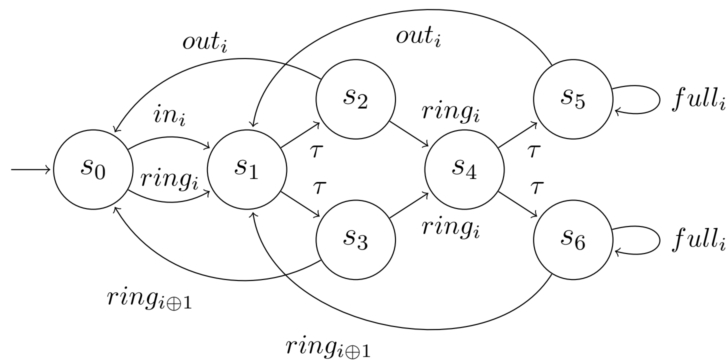

Figure 6.3: LTS of component L_i where $\oplus$ represents addition modulo 3.

$full_i$ is used to mark that component i is full when in states s_5 or s_6; it is also so when in s_4 but we do not annotate this state with this event for presentation purposes.

We want to ensure that all components can never be simultaneously full. This can be checked using static property $(\textbf{And}\,\{\textbf{Event}\,full_0, \textbf{Event}\,full_1, \textbf{Event}\,full_2\}, \models)$. Our suffix-based reachability approximation $reach_S$ (presented in Section 4.2.2.1) can show that the bad states combining states s_5 or s_6 of each component, such as (s_6, s_5, s_6), cannot be reached by finding the following inconsistency. To reach this state, component i must have performed its last event after component $i \oplus 1$'s last event. So, by going around the ring, we can derive a contradiction. ■

The third example shows how our approximations based on token invariants can be used in the analysis of a token-based system.

Example 6.3. This example describes a version of Milner's scheduler where once a component holds a token, instead of working, it accesses its critical section. So, a token rotates amongst components disposed in a ring-like topology and the holder of the token can enter its critical section. This system is captured by machine $\mathcal{S} = (\langle L_0, L_1, L_2\rangle, \mathcal{R})$ where L_0, L_1 and L_2 are defined in Figure 6.4 and $\mathcal{R}$ is the set of rules that require components to synchronise on shared events. Component L_0 has the token initially, and events tk_i represent the passage of a token from $L_{i\ominus 1}$ to L_i, where $\ominus$ is subtraction modulo 3.

To ensure that no two components enter their critical sections at the same time, we check static property $(\textbf{And}\,\{\textbf{Or}\,\{\textbf{Event}\,cs_i, \textbf{Event}\,cs_j\} \mid i \in \{0,1,2\} \wedge i \neq j\}, \models)$. Note this is an application of our strategy to capture mutual exclusion presented in Section 6.2.3. Our token-invariant approximations $reach_{C_{\mathbb{B}}}$ and $reach_C^{ce}$ (presented in Sections 5.2.1 and 5.2.3, respectively) can both capture that tokens are conserved by this system; a transition might result in tokens being exchanged between components but they cannot be created or destroyed. So, the system must have a single token at all times. They can prove that any system state involving two of component states s_0 of L_0, s_1 of L_1 and s_1 of L_2 is unreachable. Each component holds a token at these states so any combination involving two or more of these states means that the system

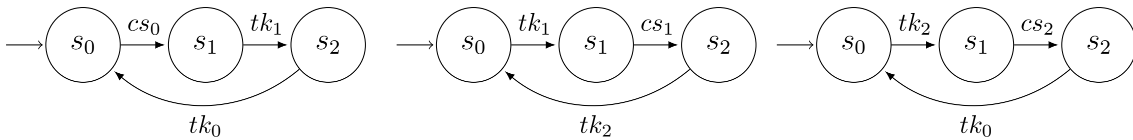

Figure 6.4: LTSs of components L_0, L_1, and L_2, respectively.

has more than one token. These combinations violate the invariant that a single token must be present in the system at all times and so they must be unreachable. ■

Even though we use a conjunction of all our approximations and check for a more general class of properties, we end up with a problem that is roughly as difficult as checking deadlock freedom. The following results justify our use of SAT and SMT solving to tackle these problems. We show later that, despite their rather elevated complexity, they can be reasonably efficiently tackled by modern solvers.

Theorem 6.13. *For a global static property, the problem of deciding whether a supercombinator machine has a system state satisfying* $candidate_{SAT}$ *is* NP*-complete, whereas for* $candidate_{SMT}$*, it is* NP*-hard. For a local property, both these problems are* NP*-hard.*

Proof. Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be the supercombinator machine we are analysing, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and $(\mathcal{V}, \sim)$ where $\sim \in \{\models, |\!\models\}$ the static property.

Firstly, we show that, for a *global* static property, (i) the problem of detecting a $candidate_{SAT}$ state belongs to *NP* and that (ii) both $candidate_{SAT}$ and $candidate_{SMT}$ problems are *NP*-hard.

To prove (i), we show that for for a fixed system state s, $candidate_{SAT}(s)$ can be decided in polynomial time. We can check each reachability approximation in $reach_{SAT}(s)$ in polynomial time[1]; see Lemmas 3.9, 4.16, 4.12, 4.8, and Theorem 5.17. So, we only need to show that $s \models \mathcal{V}$ can be decided in polynomial time. We can check whether a rule can be applied to a state in time $\mathcal{O}(\sum_{i \in \{1 \ldots n\}} |L_i|)$. Hence, we can decide which events are available and which are not in time $\mathcal{O}(|\mathcal{R}| \cdot (\sum_{i \in \{1 \ldots n\}} |L_i|))$. We can use this information to evaluate $\mathcal{V}$. The propositional atom $\mathbf{Event}\, ev$ is *true* if event ev is available, and $false$, otherwise. This evaluation can be carried out in linear time on the size of the formula.

Statement (ii) follows from Collorary 3.20. It states that the problem of checking deadlock freedom using a reachability approximation better than $reach_2$ is *NP*-complete. So, we can trivially and polynomially reduce this problem to checking static property $(\mathcal{V}_1, \models)$, which captures deadlock freedom.

For a *local* static property, we show that the corresponding problems are *NP*-hard. We present a reduction from checking a global static property to the verification of a local one. This reduction relies on the modified system $\mathcal{S}' = (\langle L'_1, \ldots, L'_n \rangle, \mathcal{R}')$ where:

[1] All reachability approximations in $reach_{SMT}$ except for $reach_D^{CA}$ can be checked in polynomial time. Because of this exception, we are unable to prove the $candidate_{SMT}$ problem belongs to *NP*.

- $L'_i = (S_i, \Sigma_i \cup \{c_i\}, \Delta_i \cup \{s, c_i, s \mid s \in S_i\}, \hat{s}_i)$
- $\mathcal{R}' = \mathcal{R} \cup \{([i, c_i], c_i) \mid i \in \{1 \ldots n\}\}$
- c_i are fresh system events.

Our reduction combines this machine with the violation sub-formula $GLOBAL \mathrel{\widehat{=}} \textbf{And}\,\{\textbf{Event}\, c_i \mid i \in \{1 \ldots n\}\}$. Checking whether $\mathcal{S}$ satisfies $(\mathcal{V}, \models)$ amounts to verifying whether $\mathcal{S}'$ satisfies $(\textbf{And}\,\{\mathcal{V}, GLOBAL\}, |\!\models)$. In our modified system, components can always individually engage on c_i triggering a top-level c_i event. So, we have that, when analysing our local satisfiability relation, $s \mathrel{|\!\models_{ss}} \textbf{Event}\, c_i$ holds *iff* i is in ss. This means that $s \mathrel{|\!\models_{ss}} \textbf{And}\,\{\textbf{Event}\, c_i \mid i \in \{1 \ldots n\}\}$ holds whenever $ss = \{1 \ldots n\}$. Therefore, by conjoining the original violation formula $\mathcal{V}$ with $GLOBAL$, we ensure that $s \mathrel{|\!\models} \textbf{And}\,\{\mathcal{V}, GLOBAL\}$ holds *iff* $s \models \mathcal{V}$ does. The construction of our modified system and violation formula clearly takes polynomial time on the input system and formula. □

6.3.2 Static-property checking via SAT/SMT solving

We built upon our SAT/SMT-checking approach proposed for detecting deadlock candidates to create an efficient implementation for StaticProperty. We implement our framework using SAT formula SP_{SAT} and SMT formula SP_{SMT}, SP_x captures system states that satisfy $candidate_x$ (for x is SAT or SMT). So, we encode the search for a violation 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 where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and $ce, ce' \in S$ two counter-example states.

$$SP_x \mathrel{\widehat{=}} State \wedge Reach_x \wedge Violation$$

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 candidate. We reuse sub-formula $State$ defined in Section 3.3.3. To implement $Reach_{SAT}$ and $Reach_{SMT}$, we reuse the propositional formulas presented in previous chapters. $Reach_2$ is defined in Section 3.3.3; $Reach_D$, $Reach_D^{DA}$, $Reach_D$, $Reach_D^{DA}$ $Reach_D^{CA}$, $Reach_S$ and $Reach_S^{DA}$ are defined in Section 4.3.2; and $Reach_{C_\mathcal{B}}$, $Reach_{E_\mathcal{B}}$, $Reach_C^{ce}$ and $Reach_{E_\mathcal{B}}^{ce'}$ are defined in Section 5.3.2. Each of these formulas ensure that the component states assigned to true in a satisfying assignment pass the corresponding reachability test.

$$Reach_{SAT} \mathrel{\widehat{=}} Reach_2 \wedge Reach_R \wedge Reach_R^{DA} \wedge Reach_S \wedge Reach_S^{DA} \wedge Reach_{C_{\mathcal{B}}} \wedge Reach_{E_{\mathcal{B}}}$$

$$Reach_{SMT} \mathrel{\widehat{=}} Reach_2 \wedge Reach_D \wedge Reach_D^{DA} \wedge Reach_D^{CA} \wedge Reach_S \wedge Reach_S^{DA} \wedge Reach_C^{ce} \wedge Reach_{E_{\mathcal{B}}}^{ce'}$$

The sub-formula *Violation* is encoded differently depending on whether the static property to be checked is global or local. For global static property $(\mathcal{V}, \models)$, *Violation* captures that the state currently assigned to true satisfies $s \models \mathcal{V}$. To encode $s \models \mathcal{V}$, we only need to introduce a way to encode $s \models \textbf{Event}\ ev$. Given an encoding for $s \models \textbf{Event}\ ev$, the satisfiability of $s \models \mathcal{V}$ for any violation formula $\mathcal{V}$ is trivially encoded based on its propositional structure.

To encode $s \models \textbf{Event}\ ev$, we need to encode the events available in a system state and, consequently, system rules. We use variable V_{ev}^i to encode that component i is in a state in which it can perform ev.

$$V_{ev}^i \Leftrightarrow \bigvee_{(s,ev',s') \in \Delta_i \wedge ev' = ev} st_{i,s}$$

Then, we capture that rule $r = ((e_1, \ldots, e_n), ev)$ can be applied by variable V_r. This variable holds whenever the system is in a state where components can perform their corresponding events, set in $(e_1, \ldots, e_n)$.

$$V_r \Leftrightarrow \bigwedge_{i \in \{1 \ldots n\} \wedge e_i \neq -} V_{e_i}^i$$

Variable $V_{\textbf{Event}\ ev}$ holds for states in which the system can perform ev, namely, whenever a rule triggering event ev can be applied. r_{ev} denotes the system event performed by rule r. Finally, the sub-formula *Violation* is then constructed by replacing **Event** ev in $\mathcal{V}$ by $V_{\textbf{Event}\ ev}$.

$$V_{\textbf{Event}\ ev} \Leftrightarrow \bigvee_{r \in \mathcal{R} \wedge r_{ev} = ev} V_r$$

For local static property $(\mathcal{V}, |\!\models)$, *Violation* captures that the system state s currently assigned to true satisfies $s \mathrel{|\!\models} \mathcal{V}$. We reuse the encoding for V_{ev}^i proposed for a global property. To encode quantification on subsystems, we introduce *participation variables* p_i. Variable p_i is true if and only if component i is part of the subsystem ss under analysis and add the clause $\bigvee_{i \in \{1 \ldots n\}} p_i$ to ensure that $ss \neq \emptyset$. As SAT/SMT checkers tend to efficiently handle existential quantification, they should be particularly effective in tackling the sort of subsystem quantification we use in our local properties.

The encoding of rule variables V_r has to take into account subsystem projections. So, rule $r = ((e_1, \ldots, e_n), ev)$ can be fired with respect to the system projection on the subsystem currently assigned to true (according to variables p_i) if and only if the following variable V_r holds. Note that if a component does not participate on a subsystem, its participation on rules is not required as per Definition 6.3. Hence, the disjunct $\neg p_i$.

$$V_r \Leftrightarrow \bigwedge_{i \in \{1 \ldots n\} \wedge e_i \neq -} (V^i_{e_i} \vee \neg p_i)$$

Variable $V_{\textbf{Event}\ ev}$ has to take into account the subsystem projections as well. For rule $r = ((e_1, \ldots, e_n), ev)$, r_C represents the components participating in r (i.e. components for which $e_i \neq -$) and r_{ev} denotes the system event performed by rule r, namely, ev. We encode variable $ron_r \Leftrightarrow \bigvee_{i \in r_C} p_i$ to represent whether rule r is on or off for the assigned subsystem. Rules that do not involve the participation of any component in this subsystem must be disregarded, that is, they are off. So, event ev can only be performed by on rules.

$$V_{\textbf{Event}\ ev} \Leftrightarrow \bigvee_{r \in \mathcal{R} \wedge r_{ev} = ev} (V_r \wedge ron_r)$$

As SP_{SAT} and SP_{SMT} captures candidate violations, if it is unsatisfiable, the system is candidate-violation free and, therefore, must satisfy the static property being checked. Otherwise, the solver returns an appropriate candidate violation.

6.3.3 Practical evaluation

Building on DeadlOx, we implemented StaticProperty in our *ApprOx* tool. It relies on FDR4's ability to analyse CSP and generate supercombinator machines to create our SAT and SMT encodings, which are then checked by the Glucose 4.0 and Z3 solvers, respectively. Also, as for DeadlOx, it solves SP_{SAT} and SP_{SMT} incrementally. It starts solving the conjunction involving $State$, $Violation$ and $Reach_2$. For SP_{SAT}, the other approximations are conjoined in the following order: $Reach_{C_\mathbb{B}}$, $Reach_{E_\mathbb{B}}$, $Reach^{DA}_R$, $Reach^{DA}_S$, $Reach_R$ and $Reach_S$. For SP_{SMT}, we have the following order: $Reach^{DA}_D$, $Reach^{CA}_D$, $Reach^{DA}_S$, $Reach_D$, $Reach_S$, $Reach^{ce}_{C_\mathbb{B}}$ and $Reach^{ce'}_{E_\mathbb{B}}$. This incremental process is used to find counter-examples ce and ce' (see Section 5.3.2). ApprOx and the models used in this section are available at [AGRR18].

The following datatype in FDR4's input language captures violation formulas:

```
datatype StaticFormula = SEv.Events | STick | STau
    | SOr.Set(StaticFormula) | SAnd.Set(StaticFormula)
    | SNot.StaticFormula
```

For a visible event e, we represent **Event** e by `SEv.e`. **Event** ✓ and **Event** τ are represented by `STick` and `STau`, respectively. Formulas **And** $\mathcal{VS}$, **And** $\mathcal{VS}$ and **Not** $\mathcal{V}$ are represented by `SAnd.`$\mathcal{VS}$, `SOr.`$\mathcal{VS}$ and `SNot.`$\mathcal{V}$, respectively.

We extend FDR4's input language with two assertions to check static properties:

```
assert SYSTEM :[local]: V :[ApprOx]
assert SYSTEM :[global]: V :[ApprOx]
```

For a violation formula `V` and a system `SYSTEM` , these assertions check for local ($\mathtt{V}, |\models$) and global ($\mathtt{V}, \models$) properties, respectively, using our SP_{SAT} encoding. To use the SP_{SMT} encoding instead, one can replace `:[ApprOx]` by `:[ApprOx [smt]]`.

We carried out two experiments: the first analyses how our framework fares in checking deadlock freedom and local-deadlock freedom, while the second evaluates how it fares for the verification of mutual-exclusion, safe-invocation and other properties. We only analyse *triple-disjoint* systems since the encoding of some of our reachability predicates as SAT/SMT constraints depend on this requirement. These experiments were conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, and 8GB of RAM.

6.3.3.1 Checking deadlock and local-deadlock freedom

In the first experiment, we compare ApprOx's encodings SP_{SAT} (ApprOx) and SP_{SMT} (ApprOxSMT) to the Deadlock Checker's FSDD and CSDD, FDR4's methods (FDR, FDRc, FDRp), and D-Finder 2's three approaches (DF2) (we only present the results of the best strategy amongst DFinder 2's approaches). Note that for the exact and D-Finder 2's methods only deadlock freedom is checked, while FSDD and CSDD check a property stronger than local-deadlock freedom.

We analyse 14 triple-disjoint systems that are free of local deadlocks. We split our results into two parts. Table 6.1 presents the first part of our results. In this part, we analyse a central lock system (Lock), a grid network implementing a simplified implementation of Raymond's algorithm (Ray) [Ray89], a grid network implementing Tarry's algorithm (Tarry) [Tar95], the mad-postman routing network (Rout), the butler solution to the dining philosophers problem (But), a distributed database (DDB), a matrix multiplication system (Mat).

	N	Approximate							Exact		
		ApprOx		**ApprOxSMT**		CSDD	FSDD	DF2	FDR	FDRc	FDRp
		Global	**Local**	**Global**	**Local**						
Lock	50	0.62	0.11	0.22	0.21	0.20	0.33	*	0.61	+	0.11
	100	0.16	0.16	0.46	0.82	0.33	0.33	*	0.11	+	0.41
	200	0.41	0.47	2.92	5.12	0.73	0.78	*	0.16	+	3.72
	500	2.87	3.42	44.35	74.81	4.59	4.74	*	0.41	+	100.62
Ray	10	0.06	0.06	0.16	0.21	0.15	0.18	*	0.11	+	0.11
	25	0.11	0.16	0.56	0.97	0.23	0.20	*	49.50	+	18.54
	50	0.16	0.26	2.42	5.27	0.23	0.28	*	*	+	*
	100	0.57	0.77	16.59	31.62	0.33	0.38	*	*	+	*
Tarry	5	0.77	0.06	0.16	0.06	-	-	3.67	0.06	+	0.06
	8	0.11	-	0.77	1.37	-	-	*	0.12	+	0.11
	12	40.45	-	*	-	-	-	*	0.36	+	0.51
	15	*	-	*	-	-	-	*	39.04	+	48.65
Rout	5	0.11	0.11	0.11	0.16	0.18	0.18	4.77	*	0.21	*
	10	0.22	0.16	0.22	0.41	0.38	0.38	*	*	0.77	*
	20	0.67	1.22	1.02	2.52	1.23	1.28	*	*	4.32	*
	30	2.32	6.63	3.62	10.33	3.43	3.69	*	*	16.02	*
But	50	-	-	1.32	1.42	-	-	*	*	*	*
	100	-	-	5.47	7.63	-	-	*	*	*	*
	150	-	-	20.10	22.10	-	-	*	*	*	*
	200	-	-	38.58	42.94	*	*	*	*	*	*
DDB	5	0.21	0.26	0.31	0.41	-	-	*	0.21	0.16	0.21
	10	3.27	3.37	5.17	8.03	*	*	*	*	*	*
	15	25.21	24.87	56.52	86.32	*	*	*	*	*	*
	20	114.04	117.35	*	*	*	*	*	*	*	*
Mat	5	0.16	0.21	0.17	0.21	-	0.18	7.13	*	0.21	0.11
	10	2.12	5.22	1.37	2.52	-	0.28	*	*	15.64	0.31
	20	24.61	*	17.50	43.65	-	0.68	*	*	*	22.65
	30	166.30	*	136.08	*	-	2.38	*	*	*	*

Table 6.1: Results for local-deadlock and deadlock freedom comparison. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify 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 (local-)deadlock freedom. + means that no efficient compression technique could be found.

Table 6.2 presents the second part of our result. In this part, we analyse a central priority-queue-lock system (PQ), a priority-based token ring system (RingP), a non-fillable ring (Ring), Milner's scheduler (Sched), a train-track system (Track), a token ring (Ring2), and a token-based message-passing grid system (MsgGrid).

ApprOx's SP_{SAT} encoding can prove all systems except for PQ and But deadlock

		Approximate							Exact		
	N	**ApprOx**		**ApprOxSMT**		CSDD	FSDD	DF2	FDR	FDRc	FDRp
		Global	**Local**	**Global**	**Local**						
PQ	3	-	-	0.26	0.26	-	-	*	0.11	+	0.06
	5	-	-	7.73	8.73	-	-	*	0.51	+	0.87
	8	-	-	*	*	-	-	*	*	+	*
	10	-	-	*	*	-	-	*	*	+	*
RingP	5	0.26	0.11	0.16	0.16	-	-	*	0.26	+	0.21
	10	0.62	0.61	1.87	1.97	-	-	*	*	+	*
	15	3.62	3.62	19.85	23.86	-	-	*	*	+	*
	20	14.59	14.44	110.33	132.67	*	*	*	*	+	*
Ring	100	0.21	0.21	0.31	0.56	0.28	-	7.03	*	0.67	*
	200	0.37	0.41	0.51	1.47	0.33	-	16.05	*	1.52	*
	300	0.62	0.72	0.82	2.42	0.53	-	40.19	*	2.72	*
	400	0.97	1.22	1.12	3.52	0.68	-	97.10	*	4.22	*
Sched	100	0.11	0.11	0.16	0.31	-	0.18	6.07	*	0.66	0.46
	200	0.21	0.21	0.26	0.67	-	0.28	9.73	*	1.62	3.72
	300	0.26	0.31	0.36	1.02	-	0.33	17.35	*	2.97	15.79
	400	0.37	0.41	0.51	1.52	-	0.43	47.01	*	4.92	45.38
Track	100	0.27	0.31	1.97	6.28	-	-	*	*	20.16	63.37
	200	0.52	1.72	11.59	27.97	-	-	*	*	211.47	*
	300	0.82	2.92	56.72	56.93	-	-	*	*	*	*
	400	1.12	5.07	59.18	97.86	-	-	*	*	*	*
Ring2	50	1.17	1.37	3.97	21.15	-	-	*	*	+	*
	100	9.98	11.59	26.71	*	-	-	*	*	+	*
	150	43.02	45.35	114.89	162.75	*	*	*	*	+	*
	200	122.61	130.37	*	*	-	-	*	*	+	*
MsgGrid	20	0.32	0.11	0.21	0.31	-	-	15.44	*	+	*
	40	0.22	0.21	0.36	0.72	-	-	*	*	+	*
	60	0.31	0.36	0.61	1.17	-	-	*	*	+	*
	80	0.47	0.46	1.22	2.12	-	-	*	*	+	*

Table 6.2: Results for local-deadlock and deadlock freedom comparison. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify 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 (local-)deadlock freedom. + means that no efficient compression technique could be found.

free, while SP_{SMT} can show all of them deadlock free. PQ relies on our token-structure detection technique $reach_C^{ce}$, whereas But needs our component-invariant technique $reach_D^{CA}$. The same analysis can be drawn for local-deadlock freedom. Note that CSDD and FSDD combine to check only 6 systems. We point out that CSDD and FSDD are examples of methods that try to prove the system's ungranted-request graph is acyclic. Hence, whenever a system passes their tests, it is free of deadlocks and

local deadlocks. The difference in precision between our method and FSDD or CSDD can be justified by the fact that we exactly characterise blocked and locally-blocked states while these other methods imprecisely assume that a cycle of ungranted request characterises them. Also, the approximation that we use seems to be more precise than theirs. So, these results show that ApprOx can handle a larger class of systems while scaling similarly when compared to other imprecise frameworks. Again, D-Finder 2's invariant calculations seem to scale quite poorly for our examples, timing out for most of the instances we tested. We point out that checking local-deadlock freedom should be more difficult for systems where components are highly interconnected. The complex behavioural dependency between components in such systems complicates the kind of analysis our approximations perform. So, for instance, verifying ring-like systems should be easier than verifying grid-like systems.

Note that for the Tarry example our tool only manages to prove local-deadlock freedom for some instances. The components in this system are arranged in a 5x($N/5$) grid that uses a token mechanism to construct a spanning tree for this grid. For $N = 5$, we have a 5x1-grid network where components can only communicate with their left and right neighbours, whereas for $N = 10$, $N = 12$ and $N = 15$, we have a grid-like systems where components can communicate additionally with up and down neighbours. For $N = 5$, a pairwise analysis (carried out by $reach_{Pair}$) can keep track of the token and show that subsystems are never blocked. For the other instances, however, their added behaviour makes the identification of the underlying token structured required. The approximation $reach^{ce}_{C}$ finds conservative token structures for all instances but for $N = 12$ and $N = 15$ the invariants derived are not strong enough to prove local-deadlock freedom.

For the complete approaches, i.e. FDR4 techniques, we only evaluate deadlock-freedom, as no built-in check for local-deadlock freedom is available in FDR4 and nor is it possible to formulate one efficiently. Approximate frameworks are consistently faster than complete approaches while being able to prove deadlock freedom for almost all analysed systems. The combination of FDR4's deadlock assertion with compression techniques comes closer to our approach in terms of verification time. We point out, however, that the effective use of compression techniques requires a careful and skilful application of those, whereas our method is fully automatic. For instance, a careful analysis of Mat's structure points to a compression strategy that is very effective. Slight modifications to it, however, lead to compression strategies that are utterly inefficient. Unsurprisingly, FDR4's deadlock assertion outperforms our framework for the Lock and Tarry examples. For these systems, a single token/lock moves around

the system and only components having a token/lock are allowed to perform actions. Hence, the state spaces of these systems are fairly small.

6.3.3.2 Checking other properties

We are unaware of any other tool that checks such a general class of properties for the sort of message-passing concurrent systems we tackle. So, we only compare our framework against refinement expressions that capture the same notions; verifying them using FDR4 creates a complete/precise approach. We check these expressions using FDR4's refinement-checking engine and its combination with partial order reduction or compression techniques. For some properties, however, we could not devise refinement expressions to capture them.

We analyse 14 triple-disjoint system. Table 6.3 shows the results of analysing a central lock system (Lock), Milner's scheduler (Sched), a priority-token ring system (RingP), a central priority-queue-lock system (PQ), a token-based message-passing grid system (MsgGrid), and a simplified implementation of Raymond's algorithm (Ray), a central read-write lock system (RWMutex), and root-based initialisation system (SI). We check that the first 6 systems implement general mutual-exclusion mechanisms, while the last two implement a read-write mutual-exclusion mechanism and a safe-invocation mechanism, respectively. The SI system is designed so it passes from an initialising to a running phase via fairly loose coordination.

Table 6.4 shows the results of analysing the asymmetric solution for the dining philosophers problem (PhilP1 and PhilP2), the butler solution to this problem (But), a simple lift system (LiftP1,LiftP2, and LiftP3), and a load-balancer system (LB). We check various properties for these systems. In PhilP1, we check that philosophers cannot all be eating at the same moment, whereas in PhilP2 we ensure that adjacent philosophers cannot be eating at the same time. In But, we verify that a philosopher must be waiting to sit at the table at all times. In LiftP1, we ensure that two lifts cannot be open on the same floor. In LiftP2, we verify that when Lift 0 reaches a floor and no other lift is open at this floor it can open the floor's door, whereas LiftP3 ensures that a lift can only open the floor's door if it is indeed at this floor. In LB, we check that all nodes cannot be working at the same time.

ApprOx is able to verify for all systems their associated properties. Since the verification of But requires our component-based-abstraction ($reach_D^{CA}$), SP_{SMT} can handle it but SP_{SAT} cannot. Our implementation offers a more scalable approach if compared to the verification of the proposed refinement expression, at least for the systems tested, since it exceeds the timeout set for fewer instances of the systems

		Approximate		Exact		
	N	**ApprOx**	**AppOxSMT**	FDR	FDRc	FDRp
Lock	50	0.11	0.36	0.16	+	2.82
	100	0.36	3.97	0.41	+	81.45
	200	17.14	28.67	1.67	+	*
	500	*	*	17.30	+	*
Sched	100	0.42	50.66	*	+	137.94
	200	1.87	*	*	+	*
	300	6.53	*	*	+	*
	400	16.05	*	*	+	*
RingP	5	0.11	0.16	0.21	+	0.26
	10	0.62	1.67	*	+	*
	15	3.52	14.39	*	+	*
	20	14.34	96.45	*	+	*
PQ	3	0.06	0.11	0.12	+	0.11
	5	0.16	2.67	0.42	+	1.32
	8	1.82	137.24	*	+	*
	10	9.03	*	*	+	*
MsgGrid	20	0.57	0.26	*	*	*
	40	0.21	0.66	*	*	*
	60	0.32	1.87	*	*	*
	80	0.41	5.12	*	*	*
Ray	10	0.26	0.21	0.26	+	0.11
	25	0.27	1.12	51.50	+	187.25
	50	1.52	11.28	*	+	*
	100	15.64	156.22	*	+	*
RWMutex	5	0.52	0.17	0.26	+	0.06
	10	0.37	7.58	0.36	+	0.87
	15	*	*	51.51	+	264.27
	20	*	*	*	+	*
SI	20	0.77	0.62	*	+	*
	40	2.82	5.52	*	+	*
	60	9.83	42.80	*	+	*
	80	34.13	68.95	*	+	*

Table 6.3: Results for other properties comparison. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the property being tested. + means that no efficient compression technique could be found.

tested. These examples demonstrate the versatility of our notation for capturing various properties and of our approximations in verifying them all.

	N	Approximate		Exact		
		ApprOx	**AppOxSMT**	FDR	FDRc	FDRp
PhilP1	50	0.11	0.16	*	1.52	*
	100	0.22	0.26	*	13.34	*
	200	0.99	0.57	*	221.17	*
	500	1.82	2.02	*	*	*
PhilP2	50	0.32	0.21	*	1.57	*
	100	0.16	0.36	*	13.24	*
	200	0.57	1.02	*	222.03	*
	500	1.87	5.72	*	*	*
But	50	-	1.72	*	+	*
	100	-	*	*	+	*
	150	-	*	*	+	*
	200	-	*	*	+	*
LiftP1	10	0.11	0.16	6.48	+	61.05
	20	1.92	3.07	*	+	*
	30	38.29	54.92	*	+	*
	40	243.69	294.86	*	+	*
LiftP2	10	0.11	0.16	+	+	+
	20	0.37	2.27	+	+	+
	30	0.92	16.04	+	+	+
	40	2.37	53.32	+	+	+
LiftP3	10	0.11	0.21	+	+	+
	20	0.36	1.32	+	+	+
	30	1.47	29.62	+	+	+
	40	3.17	133.97	+	+	+
LB	100	0.52	2.27	*	1.47	*
	200	1.92	17.60	*	6.63	*
	300	4.72	79.97	*	19.85	*
	400	9.38	252.57	*	46.35	*

Table 6.4: Results for other properties comparison. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the property being tested. + means that we could not find a refinement expression/compression techniques to tackle the property being tested.

6.4 Conclusion

Motivated by the success of using reachability approximations for the verification of deadlock freedom, we have investigated their use in checking general static properties.

We have proposed a notation to capture properties that can be specified based on the *static* (i.e. immediate) behaviour of the system, namely, the events the system can

(or cannot) perform when in a given state. These properties can be simply encoded into our SAT/SMT-based approach. Also, they do not interfere with the system analysis our reachability approximations carry out. So, they are a natural target for the sort of approximate verification framework we propose. Local-deadlock freedom, mutual exclusion, and safe invocation are some of the properties, other than deadlock freedom, that can be conveniently described in our notation. Furthermore, it can also capture local properties, a feature overlooked by most frameworks.

We have introduced an approximate framework that can handle this general class of properties. Unlike traditional frameworks, it can effectively tackle *local* static properties. To the best of our knowledge, it is the first approximate approach to check such a general class of properties in the setting we explore. The evaluation of our implementation in ApprOx show that our framework can efficiently check static properties for some practical concurrent systems that are out of the reach of complete methods. Thus, it represents a valid alternative to cope with the state-space explosion problem. Moreover, the variety of properties and systems we can verify attests to the versatility of our notation and of our approximations.

This framework inherits the strengths but also the weaknesses of our approximations. So, it is unable to prove some static property if it depends on some mechanism that our approximations cannot capture. For instance, our techniques cannot capture invariants that are strong enough to prove that the Tarry example is free of local deadlocks. Moreover, this framework is unable to (simply) handle more intricate behavioural properties such as checking traces or failures specifications. The next chapter addresses this issue by proposing an approximate framework to check traces and failures refinement using a watchdog-based approach.

Chapter 7

Approximate refinement checking

7.1 Introduction

The previous chapter presents a framework to check static properties. They specify the expected behaviour of a system based on the immediate behaviour available at individual system states. So, for instance, they cannot naturally capture *sequences* of system states that the system is allowed (expected) to engage on. To address this issue, in this chapter, we present a framework for *refinement checking* that is powered by our reachability approximations. We implement a watchdog-based approach to verify CSP's traces and stable-failures refinement. Unlike static properties, refinement expressions can naturally capture path-related properties.

Refinement checking consists of establishing whether a given finite-state system implementation meets some specification. It can suffer from two types of state-space blow-up: the normalisation (i.e. determinisation) of the specification might create a process with exponentially many states [Ros10], or the state space of the implementation might grow exponentially with the linear increase in the number of components. In practice, while the former rarely happens, the latter, traditionally known as the state-space explosion problem, is fairly common. In this chapter, we rely on our approximations to combat this problem.

The frameworks we proposed so far try to find a candidate violation by examining individual system states and the interactions they participate on. Adapting this sort of analysis for refinement checking, however, is rather challenging because instead of examining individual system states we need to examine sequences of system states. Also, it is far from obvious whether/how refinement checking can be recognised based on the interactions available for a combination of component states. To overcome this particular challenge, our framework proposes a watchdog-based approach [GMR+03]

that roughly translates checking both CSP's traces and stable-failures refinement relations into the verification of a static property.

Our framework, called *RefinementChecking*, relies on our reachability approximations and the invariants they capture to verify refinement expressions. As for all frameworks proposed in this thesis, it either shows that a refinement expression holds or produces an inconclusive result. We point out that establishing that a refinement expression holds using SAT-based symbolic exploration alone is considerably less efficient than using traditional explicit exploration [POR12]. So, the use of approximations is a key factor in making SAT/SMT-based exploration effective. To the best of the author's knowledge, our framework is the first approximate approach to tackle refinement checking for concurrent systems. We extend ApprOx to implement this framework. By testing our implementation on some systems, we show that our framework can prove a number of interesting refinement expressions, which are far from trivial, and it can do so more efficiently when compared to complete approaches.

This chapter's outline is as follows. In Section 7.2, we present our watchdog-based approach, discuss how reachability techniques can be adapted to it, and introduce and investigate our RefinementChecking framework. Finally, in Section 7.3, we present our concluding remarks.

7.2 Approximate refinement checking

In this section, we propose a refinement-checking framework that builds on our reachability approximations and SAT/SMT checking. To make this type of verification naturally fit into our constraint-based approach, we use a watchdog-based approach to convert refinement checking into verifying a static property.

7.2.1 Watchdog-based refinement checking

We have created efficient verification frameworks by replacing exact reachability by approximations. Creating such frameworks for properties that are naturally formulated as "there is no system state s that satisfies $reachable(s)$ and $bad(s)$" is quite straightforward. One can replace the $reachable(s)$ predicate in this formulation by an over-approximation, ensuring this framework looks for bad states in the over-approximation rather than in the exact state space. Refinement checking, however, is not naturally formulated in this way, and given its traditional formulation, it is far from clear whether reachability approximations can fit in a framework for refinement checking, let alone improve its efficiency.

That being said, there exists an alternative watchdog-based formulation for refinement checking that pairs a modified specification with the implementation in a way that refinement amounts to checking whether this pairing can reach a "bad" state. Here, we adapt the watchdog-based approach in [GMR+03] to our setting. The adaptations proposed are aimed at taking the best out of the combination of the watchdog-based approach with our reachability approximations.

Refinement checking for finite-state system is traditionally carried out by normalising the specification and exploring the product space of specification and implementation. This exploration, which is guided by the implementation's behaviour, either makes sure that each state of the implementation matches some compatible state of the specification, or exhibits a behaviour of the implementation that violates the specification. We assume that specification GLTSs are *normalised*, that is, no two different states can be reached via the same trace. Any finite-state GLTS can be normalised as per [Ros10], potentially incurring an exponential state-space blow-up. Normalised specifications provide a convenient way to check refinement.

The watchdog-based approach converts the normalised specification into a watchdog component that monitors the implementation's behaviour. This component goes into some *error* state as soon as it engages on a trace that is not allowed by the specification. Unlike the approach in [POR12], we add a different error state for each combination of state and event leading to a refinement violation.

Definition 7.1. Let $\mathcal{S}_I$ be a supercombinator machine, $L_I = (S_I, \Sigma_I, \Delta_I, \hat{s}_I)$ its induced LTS, and $L_{sp} = (S_{sp}, \Sigma_{sp}, \Delta_{sp}, \hat{s}_{sp}, Accs_{sp})$ a normalised specification GLTS. Also, let $\mathcal{E}_I = \Sigma_I - \{\tau\}$. The watchdog component is given by a GLTS[1] $WD(L_{sp}, \mathcal{S}_I) = (S_w, \Sigma_{sp} \cup \Sigma_I, \Delta_w, \hat{s}_{sp}, Accs_w)$ where:

- $S_w \mathrel{\widehat{=}} S_{sp} \cup \{error_{s,a} \mid s \in S_{sp} \wedge a \in \mathcal{E}_I - initials(s)\}$;
- $\Delta_w \mathrel{\widehat{=}} \Delta_{sp} \cup \{(s, a, error_{s,a}) \mid s \in S_{sp} \wedge a \in \mathcal{E}_I - initials(s)\}$;
- $Accs_w(s) = Accs_{sp}(s)$ for $s \in S_{sp}$ and $Accs_w(s) = \{\emptyset\}$ otherwise.

With multiple (individual) error states, our reachability-approximating techniques can better understand what are the behaviours leading the watchdog to these error states, and so they can better approximate reachability. If we have a single error state that is reached by all error traces, our techniques would have to create an over-approximation that takes into account all these traces at once. That is, they would

[1]We freely use a GLTS in a definition that deals with a LTS with the natural understanding that it uses the GLTS's underlying LTS structure.

be less precise in capturing which traces/events lead to a particular error state. In this case, the approximations we calculate are likely to be too imprecise. For instance, our synchronisation-analysis techniques, which are later used to approximate the watchdog's behaviour, are likely to benefit from the use of multiple error states. They create a summary/abstraction for all traces leading to a given component state. So, by having multiple individual error states, they would create individual approximations for each of these states, each of which would capture the behaviour leading to this particular state. These individual approximations are likely to be much more precise than an approximation for a single error state that has to summarise/approximate all possible error traces at once.

Moreover, note that despite making changes to the original specification states (by possibly adding new transitions), we keep the same acceptances information for these states. This sort of construction makes sense thanks to the use of an explicit function to annotate acceptance information instead of making this information a property of the events offered/refused by the transition system, see Chapter 2 for details on this aspect. We present many examples of the watchdog construction in Section 7.2.3.

We propose a supercombinator machine that pairs our watchdog with the implementation; they synchronise on shared visible events. This pairing allows the implementation to engage on any trace it wants, as the watchdog can always agree to any of the implementation's events, but the watchdog keeps track of this trace. So, the set of reachable states for this machine roughly corresponds to the set of state pairs explored in traditional refinement-checking procedures. We point to the fact that this supercombinator machine might not be triple disjoint; it might create rules requiring the participation of two implementation components plus the watchdog. This is the only supercombinator machine in this thesis not assumed to be triple disjoint.

Definition 7.2. Let $\mathcal{S}_I = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, and $L_{sp} = (S_{sp}, \Sigma_{sp}, \Delta_{sp}, \hat{s}_{sp}, Accs_{sp})$ a specification GLTS. The watchdog supercombinator machine is given by $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I) = (\mathcal{L}, \mathcal{R}')$, where:

- $\mathcal{L} = \langle L_1, \ldots, L_n, WD(L_{sp}, \mathcal{S}_I) \rangle$;
- $\mathcal{R}' = \{((e_1, \ldots, e_n, a), a) \mid ((e_1, \ldots, e_n), a) \in \mathcal{R} \land a \neq \tau\} \cup \{((e_1, \ldots, e_n, -), \tau) \mid ((e_1, \ldots, e_n), \tau) \in \mathcal{R}\}$.

For this supercombinator machine, refinement checking amounts to ensuring that a *violation* cannot be reached. For traces refinement, a violation is a state where the watchdog is in an error state. If the watchdog reaches an error state, thanks to our

watchdog-implementation pairing, it must be the case that implementation can engage on a trace not allowed by the specification.

Theorem 7.3. *Let $\mathcal{S}_I$ be a supercombinator machine, L_I its induced LTS, L_{sp} a normalised specification GLTS with states S_{sp}, S_w the states of watchdog $WD(L_{sp}, \mathcal{S}_I)$, and $S_{error} \mathrel{\widehat=} S_w - S_{sp}$. For $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ the watchdog supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $L_{sp} \sqsubseteq_T L_I$ iff there is no state $s = (s_1, \ldots, s_n, s_w) \in S$ such that $t_violation(s)$ holds, where:*

- $t_violation(s) \mathrel{\widehat=} reachable(s) \land bad_trace(s)$
- $bad_trace(s) \mathrel{\widehat=} s_w \in S_{error}$

Proof. From our definitions, one can show that a counter-example for $L_{sp} \sqsubseteq_T L_I$ can be used to construct (i.e. prove the existence of) a state s of the watchdog machine such that $t_violation(s)$, and vice-versa. $\square$

For stable-failures refinement, a violation is a state where either the watchdog is in an error state or where an acceptance violation occurs. An acceptance violation occurs when the implementation stably offers fewer events than expected.

Theorem 7.4. *Let $\mathcal{S}_I$ be a supercombinator machine, L_I its induced LTS, $initials_L(s)$ the initials predicate for L, L_{sp} a normalised specification GLTS with states S_{sp}, S_w and $Accs_w$ the states and the minimal-acceptances-annotation function of watchdog component $WD(L_{sp}, \mathcal{S}_I)$, and $S_{error} \mathrel{\widehat=} S_w - S_{sp}$. For the watchdog supercombinator machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, $L_{sp} \sqsubseteq_F L_I$ iff there is no state $s = (s_1, \ldots, s_n, s_w) \in S$ such that $sf_violation(s)$ holds, where:*

- $sf_violation(s) \mathrel{\widehat=} reachable(s) \land (bad_trace(s) \lor bad_failure(s))$
- $bad_failure(s) \mathrel{\widehat=} s \not\xrightarrow{\tau} \land \neg \exists Acc : Accs_w(s_w) \bullet Acc \subseteq initials_{L_I}((s_1, \ldots, s_n))$

Proof. From our definitions, one can show that a counter-example for $L_{sp} \sqsubseteq_F L_I$ can be used to construct (i.e. prove the existence of) a state s of the watchdog machine such that $sf_violation(s)$, and vice-versa. $\square$

7.2.2 Approximate framework

In the traditional or watchdog-based approaches, explicit state exploration generally leads to inefficient refinement checking thanks to the state-space explosion problem. Furthermore, checking that a refinement expression holds using symbolic exploration alone seems to be considerably less efficient than using explicit exploration [POR12]. So, to tame this problem, we propose a framework that combines reachability approximations and SAT/SMT checking to our watchdog-approach.

Our framework, which we call *RefinementChecking*, combines our reachability approximations with our watchdog supercombinator machine. We have to address, though, the fact that our token-based approximations ($reach_{C_{\mathbb{B}}}$, $reach_{E_{\mathbb{B}}}$, $reach^{ce}_{C}$ and $reach^{ce'}_{E_{\mathbb{B}}}$) were designed to handle only triple-disjoint systems[2]. Their extension to handle non-triple-disjoint systems is beyond the scope of this thesis. We propose, instead, an adaptation: rather than applying them to our entire watchdog machine, we propose their application to the triple-disjoint implementation machine only.

Definition 7.5. Let $\mathcal{S}_I$ be a supercombinator machine with S_I the states of its induced LTS and $ce, ce' \in S_I$ two counter-examples states, L_{sp} a normalised specification GLTS, and $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ their watchdog supercombinator machine with $L = (S, \Sigma, \Delta, \hat{s})$ its induced LTS. Also, let $reach_{C_{\mathbb{B}}}$, $reach_{E_{\mathbb{B}}}$, $reach^{ce}_{C}$ and $reach^{ce}_{E_{\mathbb{B}}}$ be the corresponding predicates calculated with respect to implementation machine $\mathcal{S}_I$. We use predicates $reachI$ to lift these predicates to the watchdog machine. For instance, $reachI_{C_{\mathbb{B}}}(s) \widehat{=} reach_{C_{\mathbb{B}}}((s_1, \ldots, s_n))$ for $s = (s_1, \ldots, s_n, s_w) \in S$, and so on.

These predicates approximate reachability for our watchdog machine.

Theorem 7.6. *Let $\mathcal{S}_I$ be a supercombinator machine with S_I the states of its induced LTS and $ce, ce' \in S_I$ two counter-examples states, L_{sp} a normalised specification GLTS, and $\mathcal{S}_{WD}(L_{sp}, \mathcal{S})$ their watchdog supercombinator machine with $L = (S, \Sigma, \Delta, \hat{s})$ its induced LTS. For any $s = (s_1, \ldots, s_n, s_w) \in S$, if s is reachable in L then $reachI_{C_{\mathbb{B}}}(s)$, $reachI_{E_{\mathbb{B}}}(s)$, $reachI^{ce}_{C}(s)$ and $reachI^{ce'}_{E_{\mathbb{B}}}(s)$ hold.*

Proof. This theorem follows from: (i) reachability for our watchdog machine can be approximated by reachability for the implementation machine (i.e. whenever it reaches a state $(s_1, \ldots, s_n, s_w)$, the implementation must be able to reach $(s_1, \ldots, s_n)$), and (ii)

[2] These approximations are not well-defined for systems that are not triple disjoint; forcefully applying our token techniques on non-triple-disjoint systems should give rise to unsound reachability predicates. The other approximations, however, can be readily used to approximate reachability for non-triple-disjoint systems.

predicates $reachI$ approximates reachability for the implementation. While (i) holds thanks to the need for cooperation between implementation and watchdog component in our watchdog machine, (ii) follows from Lemmas 5.5, 5.7, 5.10 and 5.14. □

Building upon our previous frameworks, we create two reachability tests that conjoins our reachability approximations. So, it can tackle systems implementing a combination of the interaction mechanisms captured by our techniques. Similar to StaticProperty, we have also two versions for our RefinementChecking framework, each of which uses one of the following reachability definitions.

Definition 7.7. Let $\mathcal{S}_I$ be a supercombinator machine with S_I the states of its induced LTS and $ce, ce' \in S_I$ two counter-examples states, L_{sp} a normalised specification GLTS, and $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ their watchdog supercombinator machine with $L = (S, \Sigma, \Delta, \hat{s})$ its induced LTS. For a state $s \in S$ and counter-examples $ce, ce' \in S$, we define:

$$reach_{SAT}(s) \mathrel{\widehat{=}} reach_2(s) \wedge reach_R(s) \wedge reach_R^{DA}(s) \wedge reach_S(s) \wedge reach_S^{DA}(s) \wedge reachI_{C_{\mathbb{B}}}(s) \wedge reachI_{E_{\mathbb{B}}}(s)$$

$$reach_{SMT}(s) \mathrel{\widehat{=}} reach_2(s) \wedge reach_D(s) \wedge reach_D^{DA}(s) \wedge reach_D^{CA}(s) \wedge reach_S(s) \wedge reach_S^{DA}(s) \wedge reachI_C^{ce}(s) \wedge reachI_{E_{\mathbb{B}}}^{ce'}(s)$$

It follows that since each predicate over-approximates reachability, their conjunction must also over-approximates the set of reachable states for our watchdog machine.

Theorem 7.8. *For system state s, $reachable(s)$ implies $reach_{SAT}(s)$ and $reach_{SMT}(s)$.*

As for StaticProperty, these reachability tests are still imprecise as a state might pass them and yet be unreachable. Our framework looks for a *candidate violation*, namely, a violating state that passes reachability tests $reach_{SAT}$ or $reach_{SMT}$.

Definition 7.9. Let $\mathcal{S}_I$ be a supercombinator machine L_{sp} a normalised specification GLTS with states S_{sp}, S_w the states of watchdog $WD(L_{sp}, \mathcal{S}_I)$, and $S_{error} \mathrel{\widehat{=}} S_w - S_{sp}$. For $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ the watchdog supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS, and x either SAT or SMT. For a state $s \in S$, we have the following traces and stable-failures candidate violations, respectively.

$$t_candidate_x(s) \mathrel{\widehat{=}} reach_x(s) \wedge bad_trace(s)$$

$$sf_candidate_x(s) \mathrel{\widehat{=}} reach_x(s) \wedge (bad_trace(s) \vee bad_failure(s))$$

Furthermore, as our framework over-approximates reachability, it is sound in the sense that if no candidates are found, the corresponding refinement relation must hold.

Theorem 7.10. *For a supercombinator machine $\mathcal{S}_I$, a normalised specification GLTS L_{sp} and x either SAT or SMT, $L_{sp} \sqsubseteq_T L_I$ holds if there is no state s of their watchdog machine such that $t_candidate_x(s)$ holds. Similarly, $L_{sp} \sqsubseteq_F L_I$ holds if there is no state s such that $sf_candidate_x(s)$ holds.*

7.2.3 Precision and complexity of RefinementChecking

Similar to StaticProperty, our refinement-checking framework inherit the strengths and weaknesses of our approximations. Unlike it, however, this new framework has to approximate reachability for a more complex machine. Our approximations have to approximate the behaviour of the implementation machine – this part is similar to what we do for verifying static properties and checking deadlock freedom – but they also have to approximate how components in the implementation machine interact with the watchdog component. This additional step makes our analyses for refinement checking more intricate than they are for static properties.

We present five examples that illustrate how our approximations take part in refinement checking. The first example demonstrates the use of pairwise analysis.

Example 7.1. The implementation system is a version of Milner's scheduler. In this system, a token is passed around a ring of components and the component possessing the token can work. This system is captured by machine $\mathcal{S}_I = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ where L_0, L_1, and L_2 are described in Figure 7.1, and $\mathcal{R}$ requires components to synchronise on shared events; the rules synchronising events tk_i propagate this event as the machine event whereas the others trigger a machine-level τ event. In Figure 7.2, we present the normalised specification GLTS L_{sp} where the faded additions form the watchdog component $WD(L_{sp}, \mathcal{S}_I)$. Note the error states of this component are $s_3, \ldots, s_8$. The refinement expression $L_{sp} \sqsubseteq_T L_I$, where L_I is the LTS induced by $\mathcal{S}_I$, captures that the token rotates amongst components in the implementation machine.

Pairwise analysis can show that this refinement holds. Our predicate $reach_2$ over our watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ can show that no reachable state of this machine can include a watchdog component's error state. It is sufficient to examine the behaviour of each implementation component paired with the watchdog component to show that. Each implementation component requires a pair of tk_i events to alternate. For instance, L_0 requires the alternation of tk_1 then tk_0. Thus, when we analyse the behaviour of L_0 paired with the watchdog component, we can derive that error states s_3, s_6 and s_7 are unreachable as the watchdog component can only reach these error states by performing traces that do not respect this alternation. The same argument

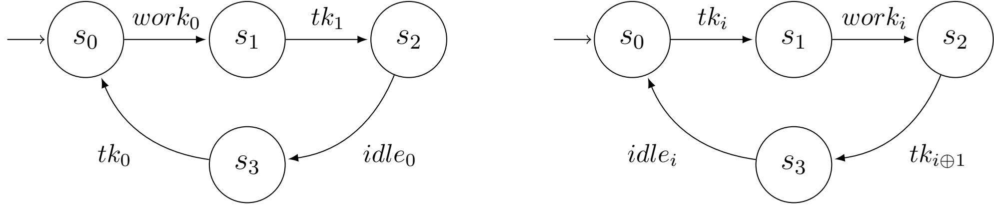

Figure 7.1: LTSs L_0 and L_i for $i \in \{1, 2\}$, respectively.

shows that L_1's behaviour makes states s_4, s_6 and s_8 of the watchdog component unreachable, whereas L_2 shows unreachable s_3, s_5, s_8. Hence, we have that all error states are unreachable and the refinement must hold. This analysis can prove this token-rotation property for similar rings with more than 3 components, as well. ■

The second example shows how a combination of pairwise and synchronisation analyses can show that a ring system respects an input-output behaviour.

Example 7.2. This example presents an abstract configuration-ring system. A configuration component chooses which configuration message is going to be sent around the ring. This configuration message rotates around the ring, alerting components in the ring for the new configuration chosen, until it reaches back the configuration component. At this point, the configuration component can choose another configuration message to send around the ring.

This system is implemented by machine $\mathcal{S}_I = (\langle L_0, L_1, L_2, L_3 \rangle, \mathcal{R})$ where L_0, L_1, L_2 and L_3 are described in Figure 7.3, and $\mathcal{R}$ requires components to synchronise on shared events; the rules synchronising events $c_{3,0}$, $c_{3,1}$, $c_{0,0}$, and $c_{0,1}$ propagate these events as the machine event whereas the others trigger a machine-level τ event. An arrow with two labels is a shorthand for two transitions with the same source and target states but with different labels. L_3 is the configuration component in this system, whereas the others are simple members of the ring which listen to and pass the

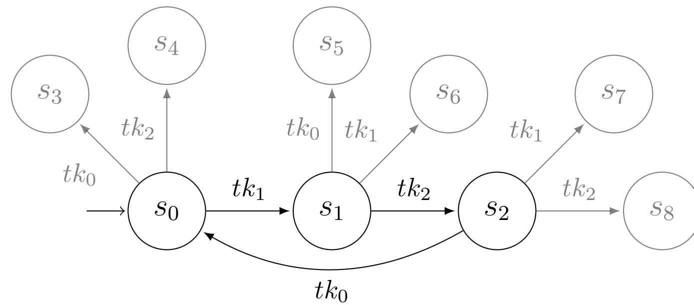

Figure 7.2: GLTS L_{sp} with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$.

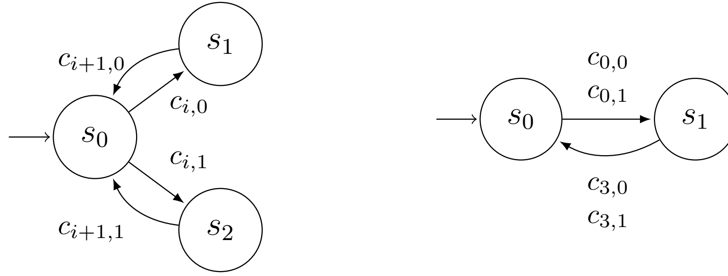

Figure 7.3: LTSs L_i for $i \in \{0, 1, 2\}$ and L_3, respectively.

configuration message. Events $c_{i,0}$ ($c_{i,1}$) depicts that configuration 0 (1) was chosen. Event $c_{i,v}$ represents the passing of configuration message v from component $L_{i \ominus 1}$ to L_i. Figure 7.4 presents the normalised specification GLTS L_{sp} and the watchdog component $WD(L_{sp}, \mathcal{S}_I)$ where $s_3, \ldots, s_{10}$ are error states. The refinement expression $L_{sp} \sqsubseteq_T L_I$, where L_I is the LTS induced by $\mathcal{S}_I$, captures that the message that gets back to L_3 matches the configuration it has previously chosen.

A combination of pairwise analysis and synchronisation analysis can show that this refinement expression holds. Predicates $reach_2$ and $reach_R$ over our watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ can show that no reachable state of this machine can include a watchdog component's error state. Component L_3 requires the alternation of events $c_{0,0}$ or $c_{0,1}$ then $c_{3,0}$ or $c_{3,1}$ but the watchdog component can only reach error states s_7 and s_{10} by performing traces that respect this alternation. So, the pairwise analysis of L_3 and the watchdog component identifies this cooperation incompatibility and shows that all error states except s_7 and s_{10} are unreachable. Error states s_7 and s_{10} can be shown unreachable by $reach_R$. This technique automatically captures the following argument. For state s_7, the analysis of the watchdog shows that the number of $c_{3,1}$ events performed to reach this state must be higher than the number of events $c_{0,1}$. The analysis of the implementation machine, however, shows that it can only reach states where either events $c_{3,1}$ and $c_{0,1}$ have been performed the same number of times or $c_{0,1}$ has been performed more times than $c_{3,1}$. This contradiction shows that this error state must be unreachable. A symmetric argument can be drawn for state s_{10} and events $c_{3,0}$ and $c_{0,0}$. The analysis we present here can also ensure this input-output behaviour for similar rings with more than 4 components. ■

The third example illustrates how a combination of pairwise and synchronisation analyses can show a counting-based property.

Example 7.3. This example presents a version of the well-known asymmetric solution to the dining philosophers problem. In this system, $N = 4$ philosophers compete

to acquire a pair of forks in order to eat. Each philosopher, sat on a round table, acquires the fork positioned on its left-hand side and another on its right-hand side. All philosophers acquire first the left-hand-side fork and then the right-hand-side one, except for one of them which acquires the forks in the opposite order. This system is captured by machine $\mathcal{S}_I = (\langle L_0, \ldots, L_7 \rangle, \mathcal{R})$ such that $L_0, \ldots, L_3$ are Philosophers $0, \ldots, 3$ as described in Figure 7.5 and $L_4, \ldots, L_7$ are Forks $0, \ldots, 3$ as per Figure 7.6, and $\mathcal{R}$ requires components to synchronise on shared events; the rules synchronising events $done_i$ and events eat_i propagate them as the machine event whereas the others trigger a machine-level τ event. Events $picksup_{i,j}$ and $putsdown_{i,j}$ denote the picking up and putting down of Fork j by Philosopher i, whereas $wait_i$, eat_i and $done_i$ denote the individual activities of Philosopher i. Figure 7.6 depicts the normalised specification GLTS L_{sp} and the watchdog component $WD(L_{sp}, \mathcal{S}_I)$; its error states are $s_3, \ldots, s_{10}$. Transitions annotated with eat_* and $done_*$ compactly represent a set of transitions involving all events eat_i and $done_i$, respectively. The refinement expression $L_{sp} \sqsubseteq_T L_I$, where L_I is the LTS induced by $\mathcal{S}_I$, captures that at most two philosophers can be eating simultaneously.

A combination of pairwise analysis and synchronisation analysis can show that this refinement expression holds. Predicates $reach_2$ and $reach_D^{CA}$ over our watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ can show that no reachable state of this machine can include a watchdog component's error state. Let $\#done$ and $\#eat$ count the number of times events $done_i$ and eat_i must have been performed to reach the system state in

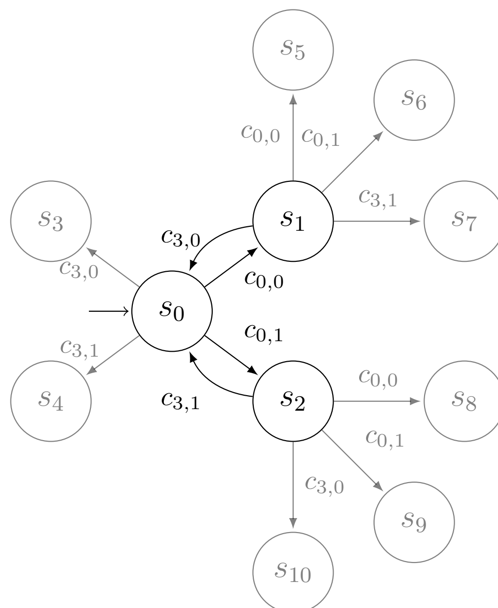

Figure 7.4: GLTS L_{sp} with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$.

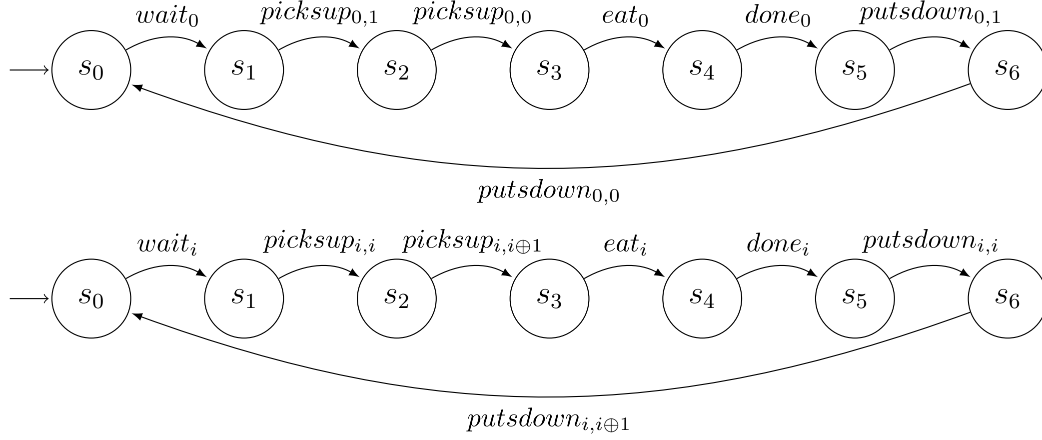

Figure 7.5: LTSs of Philosophers 0 and i for $i \in \{1, 2, 3\}$, where $\oplus$ represents subtraction modulo 4, respectively.

discussion, respectively. Our $reach_D^{CA}$ technique can deduce that for every system state of the implementation machine either $\#eat > \#done$ or $\#eat = \#done$. This fact alone shows that error states $s_3, \ldots, s_6$ are unreachable because the watchdog component requires $\#eat < \#done$ to reach these states. For error states $s_7, \ldots, s_{10}$, this fact is not enough so we need to rely on our pairwise analysis to prove them unreachable. Pairwise analysis can show that the implementation machine can only reach states where at most two philosophers are eating. Hence, its combination with $reach_D^{CA}$ can show that $\#eat - \#done < 3$ holds for all implementation machine states. The analysis of the watchdog by $reach_D^{CA}$, however, shows that in any of these error states $\#eat - \#done = 3$. This contradiction proves that error states $s_7, \ldots, s_{10}$ are unreachable. This analysis can show that for a similar system with $N \geq 4$ philosophers at most $N/2$ can be eating simultaneously. ■

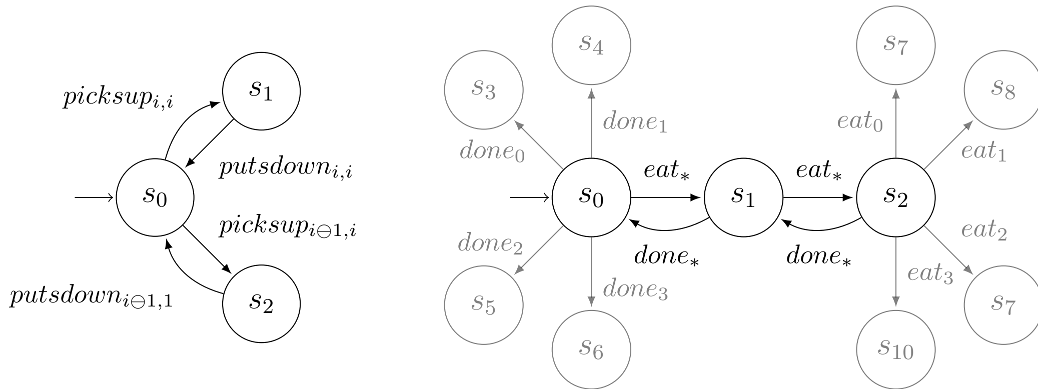

Figure 7.6: LTS of Fork i, where $\ominus$ represents subtraction modulo 4, and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively.

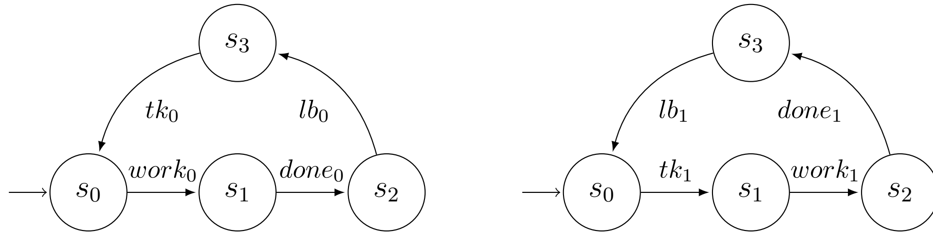

Figure 7.7: LTSs of components L_0 and L_1, respectively.

The next example shows how pairwise analysis and the detection of conservative token structures can capture a sort of mutual-exclusion property.

Example 7.4. This example relies on a token-based load-balancer implementation system. A load-balancer component passes a token to one of the worker components to activate it; the load-balancer does some processing after passing the token and then goes into an idle state where it waits for the token to be passed back to it. The activated worker works and then returns the token back to the load-balancer component. At this point, the balancer can choose again a worker to activate.

This system is captured by supercombinator machine $\mathcal{S}_I = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 7.7 and 7.8, and $\mathcal{R}$ the set of rules that require components to synchronise on shared events; the rules synchronising events $work_i$ and $done_i$ propagate them as the machine event whereas the others trigger a machine-level

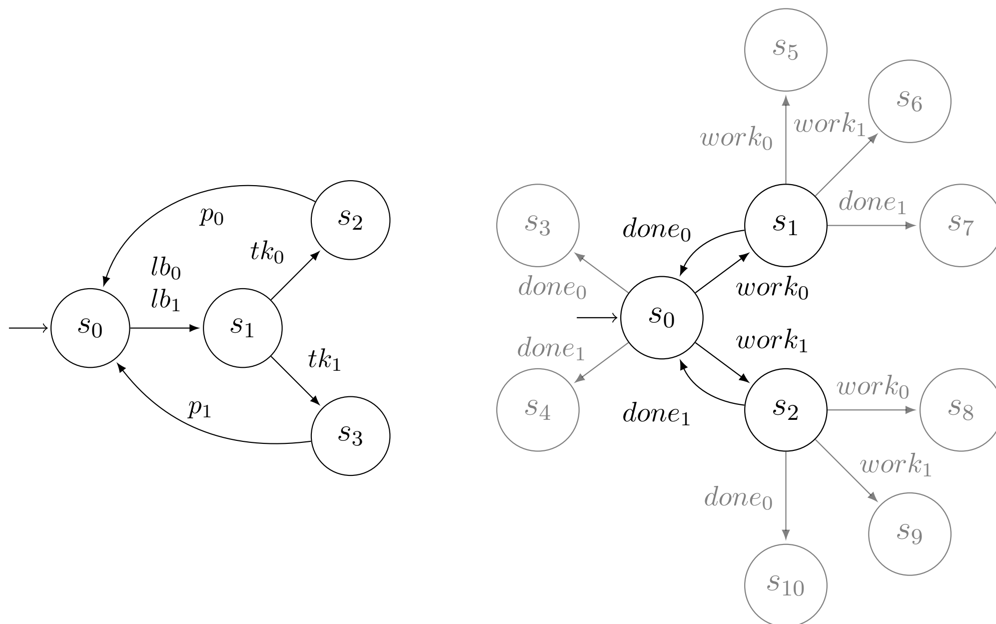

Figure 7.8: LTS of component L_2 and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively.

τ event. An arrow with two labels represents two transitions with the same source and target states but with different labels. Component L_2 is the load-balancer and L_0 and L_1 are worker components. Events tk_i are used by L_2 to give control (i.e. pass a token) to component L_i and lb_i passes the control back from L_i to the balancer L_2. Initially process L_0 is activated, i.e. has the token. Figure 7.8 presents the normalised specification GLTS L_{sp} and the watchdog component $WD(L_{sp}, \mathcal{S}_I)$; its error states are $s_3, \ldots, s_{10}$. The refinement expression $L_{sp} \sqsubseteq_T L_I$, where L_I is the LTS induced by $\mathcal{S}_I$, captures that components L_0 and L_1 cannot be active at the same time; a mutual exclusion property.

A combination of pairwise analysis and conservative-token-structure detection can show that this refinement expression holds. Predicates $reach_2$ and $reachI_{C_{\mathbb{B}}}$ can show that no reachable state of our watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ can include a watchdog component's error state. The pairwise analysis captured by $reach_2$ can show that error states $s_3, s_4, s_5, s_7, s_9, s_{10}$ are unreachable. Component L_0 requires the alternation of $work_0$ then $done_0$. By the pairwise analysis of L_0 and watchdog component $WD(L_{sp}, \mathcal{S}_I)$, we can show that error states s_3, s_5 and s_{10} are unreachable because no watchdog-component trace that can reach one of these states respect this alternation. A similar argument, based on the alternation of $work_1$ and $done_1$ ensured by L_1, can be captured by the pairwise analysis of L_1 and $WD(L_{sp}, \mathcal{S}_I)$, proving that error states s_4, s_7 and s_9 are unreachable. A combination of pairwise analysis and $reachI_{C_{\mathbb{B}}}$ can show that the two error states left, s_6 and s_8, are unreachable. Pairwise analysis ensure that these states can only be reached when both L_0 and L_1 are active, i.e. they both hold a token. We can detect with $reachI_{C_{\mathbb{B}}}$, however, that the implementation machine implements a conservative token mechanisms which has a single token initially. So, it cannot reach a state where both L_0 and L_1 hold a token, and consequently error states s_6 and s_8 cannot be reached. This analysis can show that similar systems with $N \geq 3$ components respect this property. ■

Finally, the fifth example shows how synchronisation analysis (or existential-token-structure detection) can prove deadlock freedom, formulated as refinement.

Example 7.5 (From [Ros10])**.** This example relies on a non-fillable ring as the implementation system. In this message-exchange system, components are arranged in a ring and each of them can receive messages either from its predecessor or from its user. If it holds a message, it can pass the message along to the next component in the ring or output the message to its user. Each component can hold up to two

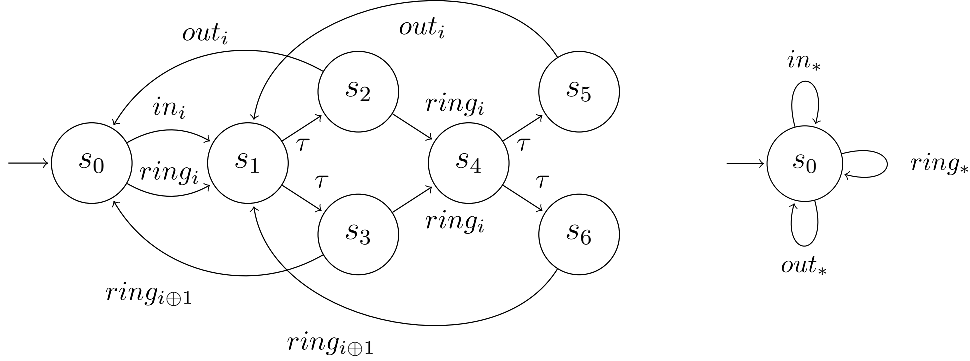

Figure 7.9: LTS of component L_i for $i \in \{0, 1, 2\}$, where $\oplus$ represents addition modulo 3, and GLTS L_{sp}, respectively.

messages at a time and the second message must have come from its predecessor in the ring and not from its user.

This system is described by machine $\mathcal{S}_I = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ with L_0, L_1 and L_2 defined in Figure 7.9 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events. A component receives messages from its predecessor via event $ring_i$, and from its user via event in_i. It can pass a message along to the next component in the ring via event $ring_{i\oplus 1}$, and output the message to its user via out_i. The τ transitions represent an internal (non-deterministic) decision of the component. Figure 7.9 also presents the normalised specification GLTS L_{sp}, where $Acc(s_0) = \{\{e\} \mid e \in \Sigma_I - \{\tau\}\}$ and Σ_I is the alphabet of the LTS induced by $\mathcal{S}_I$. Note that the watchdog component $WD(L_{sp}, \mathcal{S}_I)$ has no error state and so it coincides with L_{sp}. The refinement expression $L_{sp} \sqsubseteq_F L_I$, where L_I is the LTS induced by $\mathcal{S}_I$, captures deadlock freedom for L_I.

Synchronisation analysis can show that this refinement expression holds. Predicates $reach_S$ can show that no reachable state of our watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ can have all implementation components full, i.e. in their state s_6. Our suffix-based reachability approximation $reach_S$ shows that such a state is not reachable by finding the following inconsistency. To reach it, implementation component L_i must have performed its last event after component $i \oplus 1$'s last event. So, by going around the ring, we can derive a contradiction. A similar argument can be made using the existential token detection implemented in $reachI_{E_{\mathbb{B}}}$. This analysis can show that this refinement expression holds for similar systems with $N \geq 3$ components. ■

These examples demonstrate that our approximations are capable of proving that some common and far-from-trivial refinement expressions hold. They also show

how our approximations can leverage invariants maintained by common concurrent-systems interaction mechanisms to prove them. We have shown how pairwise analysis, synchronisation analysis, and token-structure detection can capture resource-allocation, token-based and counting mechanisms. These examples also hint to the fact that pairwise analysis plays quite an important role in analysing the possible interactions between implementation components and the watchdog component.

Nevertheless, our framework cannot handle systems that cannot be tightly approximate by our reachability tests. For instance, generally showing that a concurrent system does not allow message overtaking is beyond the capability of our framework. This sort of invariant is particularly important when proving, for example, that a system behaves as a buffer. Moreover, the use of our token-structure-detection techniques also makes our framework unpredictable in the sense that we might not be able to anticipate which token invariant will be captured.

Despite handling a different class of properties, which might involve path-based analysis, RefinementChecking's candidate-detection problems are roughly as complex as their counterparts for global static properties. Their *NP*-hardness also justifies our use of SAT/SMT solving.

Theorem 7.11. *Given a supercombinator machine $\mathcal{S}_I$ and a normalised GLTS L_{sp}, the problem of deciding whether the watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ has a system state satisfying $t_candidate_{SAT}$ is* NP-*complete. The same holds for detecting a state satisfying $sf_candidate_{SAT}$. If we consider $t_candidate_{SMT}$ and $sf_candidate_{SMT}$, both these problems are* NP-*hard.*

Proof. Given supercombinator machine $\mathcal{S}_I$ and normalised GLTS L_{sp}, we prove the following results.

Firstly, we show that the problem of detecting a $t_candidate_{SAT}$ state for the watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ (i) belongs to *NP* and (ii) is *NP*-hard.

(i) We prove this claim by showing that for a fixed system state s, $t_candidate_{SAT}(s)$ can be decided in polynomial time. $bad_trace(s)$ can be trivially checked in polynomial time by examining whether s includes an error state of the watchdog component. Since each reachability-approximating predicate forming $reach_{SAT}$ can be checked in polynomial time (see Lemmas 3.9, 4.16, 4.12, 4.8, and Theorem 5.17), their conjunction can also be checked in polynomial time.

(ii) To demonstrate this claim, we present a polynomial-time reduction from the CNF-SAT problem to the problem of detecting whether the watchdog machine

$\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ has a state s such that $bad_trace(s)$ and $reach(s)$ hold, where $reach$ is a predicate that over-approximates reachability at least as precisely as $reach_2$. Note that $reach_{SAT}$ meets this requirement.

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

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

Component F_i captures the satisfiability of clause $\mathcal{F}_i$, component X_i models the assignment of boolean variable x_i, and component CC is meant to perform event sat if the formula is satisfied. We use literals x_j and $\neg x_j$ as events that denote whether variable x_i has been assigned to $true$ and $false$, respectively. So, in particular, note $\mathcal{F}_{i,j}$ are events. Next, we present the definitions of component F_i, X_i and CC and their graphical representation in Figures 7.10 and 7.11.

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

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

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

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

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

* $S = \{s_0, \ldots, s_{n+1}\}$
* $\Sigma = \{sat_i \mid i \in \{1 \ldots n\}\} \cup \{sat\}$
* $\Delta = \{(s_{i-1}, sat_i, s_i) \mid i \in \{1 \ldots n\}\} \cup \{(s_n, sat, s_{n+1})\}$
* $\hat{s} = s_0$

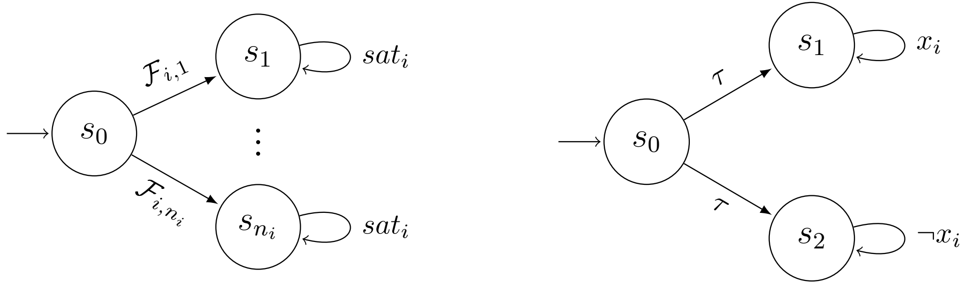

Figure 7.10: LTSs F_i and X_i respectively.

The rules we propose allow each clause component to synchronise with the variable components capturing variables that participate on this clause. Moreover, they allow clause components F_i to synchronise on sat_i with CC, and CC to perform sat by itself. All rules trigger a machine-level τ event except for the rule that allows CC to perform sat, which propagates the machine-level sat event. We use $F_i = i$, $X_{i,j} = n + m_{i,j}$ and $CC = n + m + 1$ to conveniently denote the positions of components in the event tuple, where $m_{i,j}$ denotes the index of the boolean variable in literal $\mathcal{F}_{i,j}$, so if $\mathcal{F}_{i,j} = x_k$ or $\mathcal{F}_{i,j} = \neg x_k$ then $m_{i,j} = k$.

$$\begin{aligned}\mathcal{R}' = \; & \{([CC, sat], sat)\} \cup \{([(F_i, sat_i), (CC, sat_i)], \tau) \mid i \in \{1 \dots n\}\} \\ & \cup \{([n+k, \tau], \tau) \mid k \in \{1 \dots m\}\} \\ & \cup \bigcup_{i \in \{1 \dots n\}} \{([(F_i, \mathcal{F}_{i,j}), (X_{i,j}, \mathcal{F}_{i,j})], \tau) \mid j \in \{1 \dots n_i\}\}\end{aligned}$$

In our machine, component F_i can reach a state looping on event sat_i if it has been satisfied given the current assignment captured by components X_i. Component CC reaches state s_{n+1} and performs sat if all the clauses have been satisfied by the current assignment, i.e. the formula has been satisfied.

To show that this reduction is valid we show that (a) the watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}')$, where L_{sp} is described in Figure 7.11, can be constructed in polynomial time and that (b) the original formula has a satisfying assignment *iff* we can detect a state s that satisfies $bad_trace(s) \wedge reach(s)$ for the watchdog machine. Note $reach(s)$ is a predicate that over-approximates reachability, i.e. $reachable(s) \Rightarrow reach(s)$, and it does so at least as precisely as $reach_2(s)$, namely, $reach(s) \Rightarrow reach_2(s)$ for this watchdog machine.

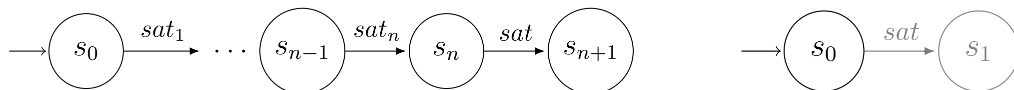

Figure 7.11: LTS CC and GLTS L_{sp} (with faded additions forming LTS $WD(L_{sp}, \mathcal{S}_I)$), respectively.

(a) We can deduce from our definitions that $\mathcal{S}'$, and consequently $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}')$, can be constructed in polynomial time on the size of boolean formula $\mathcal{F}$.

(b) If $\mathcal{F}$ has a satisfying assignment then there exists a state s that satisfies $bad_trace(s) \wedge reach_2(s)$ for the watchdog machine. We use the satisfying assignment to reach such a state. We can make the machine transition to a state where variable components X_i capture this satisfying assignment. From this state, we can make each component F_i transition to a state performing sat_i. Then, CC transition to s_n at which point it can synchronise on sat with the watchdog component $WD(L_{sp}, \mathcal{S}')$ to reach s_{n+1}, making the watchdog component reach s_1. Thus, this machine state is reachable, so $reach(s)$ holds, and satisfies $bad_trace(s)$.

If there exists a state s that satisfies $bad_trace(s) \wedge reach(s)$ for the watchdog machine then $\mathcal{F}$ has a satisfying assignment. We show that the assignment captured by components X_i in this machine state must satisfy $\mathcal{F}$. Since $reach(s) \Rightarrow reach_2(s)$, we can use pairwise reachability to construct the following argument. If $bad_trace(s)$ holds, CC must be in state s_{n+1}. So, we know that each component F_i must be in a state where it can perform sat_i, and these components can only have reached such states if one of its literals have been satisfied by the assignment captured by components X_i.

Secondly, we show that the problem of detecting a $sf_candidate_{SAT}$ state for the watchdog machine $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I)$ (iii) belongs to NP and (iv) is NP-hard.

(iii) We prove this claim by showing that for a fixed system state s, $sf_candidate_{SAT}(s)$ can be decided in polynomial time. In the proof of claim (i) we show that $bad_trace(s)$, and all predicates in $reach_{SAT}$ can be checked in polynomial time. $bad_failure(s)$ can also be checked in polynomial time on the size of the watchdog machine by examining the refusal sets associate to the state of the watchdog component in s.

(iv) This claim follows from Collorary 3.20. It states that the problem of detecting a blocked state using a reachability approximation better than $reach_2$ is NP-complete. So, we can trivially and polynomially reduce the problem of checking for a blocked state using $reach_{SAT}$ to checking for a $sf_candidate$. It suffices to use specification $L_{sp} = (\{s_0\}, \Sigma_I, \{s_0, e, s_0 \mid e \in \Sigma_I - \{\tau\}\}, s_0)$, where Σ_I is the alphabet of the implementation machine and $Acc(s_0) = \{\{e\} \mid e \in \Sigma_I - \{\tau\}\}$.

The reduction proposed to prove (ii) can be used to show that the problem of detecting a watchdog machine state that satisfies $t_candidate_{SMT}$ is NP-hard, whereas an argument similar to the one given to prove (iv) can show that the problem of detecting a $sf_candidate_{SMT}$ is NP-hard. □

7.2.4 Refinement checking via SAT/SMT solving

We built upon our SAT/SMT-checking approach to create an efficient implementation for RefinementChecking. We implement our framework using SAT formulas RCt_{SAT} and $RCsf_{SAT}$, and SMT formulas RCt_{SMT} and $RCsf_{SMT}$. While RCt_x captures watchdog machine states that satisfy $t_candidate_x$, where x is SAT or SMT, $RCsf_x$ captures $sf_candidate_x$ states. So, we encode the search for a violation candidate as a satisfiability problem to be later checked by a SAT/SMT solver. For the remainder of this section, let $\mathcal{S}_I = (\langle L_1, \ldots, L_n \rangle, \mathcal{R}_I)$ be a supercombinator machine where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$ and ce, ce' are two of its states, and $L_{sp} = (S_{sp}, \Sigma_{sp}, \Delta_{sp}, \hat{s}_{sp}, Accs_{sp})$ a normalised specification GLTS. Also, let $\mathcal{S}_{WD}(L_{sp}, \mathcal{S}_I) = (\langle L_1, \ldots, L_n, L_w \rangle, \mathcal{R})$ be the watchdog supercombinator machine under analysis, where $L_w = (S_w, \Sigma_w, \Delta_w, \hat{s}, Accs_{sp})$ is the watchdog component $WD(L_{sp}, \mathcal{S}_I)$ with error states $S_{error} \mathrel{\widehat{=}} S_w - S_{sp}$, and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS.

$$
\begin{aligned}
RCt_x &\mathrel{\widehat{=}} State \wedge Reach_x \wedge BadTrace \\
RCsf_x &\mathrel{\widehat{=}} State \wedge Reach_x \wedge (BadTrace \vee BadFailure)
\end{aligned}
$$

We use boolean variables $st_{i,s}$ to represent state s of component i. Our formulas are constructed so the combination of $st_{i,s}$ variables assigned to true in a satisfying assignment forms an appropriate candidate. We reuse sub-formula $State$ defined in Section 3.3.3. To implement $Reach_{SAT}$ and $Reach_{SMT}$, we reuse the propositional formulas presented in previous chapters. $Reach_2$ is defined in Section 3.3.3; $Reach_D$, $Reach_D^{DA}$, $Reach_D$, $Reach_D^{DA}$ $Reach_D^{CA}$, $Reach_S$ and $Reach_S^{DA}$ are defined in Section 4.3.2; and $ReachI_{C_{\mathbb{B}}}$, $ReachI_{E_{\mathbb{B}}}$, $ReachI_C^{ce}$ and $ReachI_{E_{\mathbb{B}}}^{ce'}$ are defined in Section 5.3.2, unlike the other predicates they are calculated with respect to the implementation machine $\mathcal{S}_I$. Each of these formulas ensure that the component states assigned to true in a satisfying assignment pass the corresponding reachability test.

$$
\begin{aligned}
Reach_{SAT} \mathrel{\widehat{=}} \;& Reach_2 \wedge Reach_R \wedge Reach_R^{DA} \wedge Reach_S \\
& \wedge Reach_S^{DA} \wedge ReachI_{C_{\mathbb{B}}} \wedge ReachI_{E_{\mathbb{B}}} \\
Reach_{SMT} \mathrel{\widehat{=}} \;& Reach_2 \wedge Reach_D \wedge Reach_D^{DA} \wedge Reach_D^{CA} \\
& \wedge Reach_S \wedge Reach_S^{DA} \wedge ReachI_C^{ce} \wedge ReachI_{E_{\mathbb{B}}}^{ce'}
\end{aligned}
$$

The sub-formula $BadTrace$ captures whether the predicate bad_traces holds for the watchdog machine state currently assigned to true.

$$BadTrace \mathrel{\widehat{=}} \bigvee_{s \in S_{error}} st_{w,s}$$

To encode stable-failures refinement, we need to capture the predicate $bad_failures$ and, consequently, the events that are offered by the implementation. The variable V_e^i captures that component i is in a state in which it can perform e.

$$V_e^i \Leftrightarrow \bigvee_{(s,e',s') \in \Delta_i \wedge e'=e} st_{i,s}$$

Then, we capture that rule $r = ((e_1, \ldots, e_n), a) \in \mathcal{R}_I$ can be applied by variable V_r. This variable holds whenever the implementation machine is in a state where its components can perform the event required by r. Note that we encode the implementation rules and not the watchdog machine ones.

$$V_r \Leftrightarrow \bigwedge_{i \in \{1 \ldots n\} \wedge e_i \neq -} V_{e_i}^i$$

Variable V_a holds for states in which the implementation can perform a rule triggering event a. r_{ev} denotes the system event performed by rule r.

$$V_a \Leftrightarrow \bigvee_{r \in \mathcal{R} \wedge r_{ev}=a} V_r$$

For a minimal acceptance $Acc \in \mathbb{P}(\Sigma_w)$, we use variable V_{Acc} to capture that, for the state currently assigned to true, the implementation cannot agree to Acc, namely, it refuses an event that is in Acc, or formally, $Acc \nsubseteq initials_{L_I}$ where $initials_{L_I}$ gives the initials of the implementation for the state currently assigned to true.

$$V_{Acc} \Leftrightarrow \bigvee_{a \in Acc} \neg V_a$$

For each $s_w \in S_w$, we add a constraint that makes the variable V_{accs} capture that a minimal-acceptances violation has occurred provided the watchdog component is in state s_w. If the watchdog machine is in state $(s_1, \ldots, s_n, s_w)$, such a violation occurs when the implementation cannot agree to any $Acc \in Accs_w(s_w)$.

$$st_{w,s_w} \Rightarrow (V_{accs} \Leftrightarrow \bigwedge_{Acc \in Accs_w(s_w)} V_{Acc})$$

The predicate $bad_failure$ can be, then, captured by constraint $BadFailure$. It holds whenever a stable state (that is, one for which $\neg V_\tau$ holds) produces a minimal-acceptances violation.

$$BadFailure \mathrel{\widehat{=}} \neg V_\tau \wedge V_{accs}$$

RCt_x and $RCsf_x$ capture candidate violations for our watchdog system. So, if one of these formulas is unsatisfiable, the watchdog supercombinator machine is free of candidates and, therefore, the corresponding refinement expression must hold. Otherwise, the solver returns an appropriate candidate violation.

7.2.5 Practical evaluation

We extended our ApprOx tool to implement RefinementChecking using the proposed SAT and SMT encodings. It relies on the same incremental solving process used to implement StaticProperty in Section 6.3.3. We extend FDR4's input language so one can annotate a refinement assertion with `:[ApprOx]`. For a specification process `SPEC` and system `SYSTEM`, the following two assertions are checked using ApprOx with encodings RCt_{SAT} and $RCsf_{SAT}$, respectively. To use the RCt_{SMT} and $RCsf_{SMT}$ encodings instead, one can use `:[ApprOx [smt]]`. ApprOx and the models used in this section are available at [AGRR18].

```
assert SPEC [T= SYSTEM :[ApprOx]
assert SPEC [F= SYSTEM :[ApprOx]
```

We analyse how our framework fares in checking traces and stable-failures refinement when compared to FDR4's explicit-exploration refinement-checking engine (FDR4) and its combination with partial order reduction (FDRp) or compression techniques (FDRc). Our analyses were conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, and 8GB of RAM.

7.2.5.1 Traces refinement checking

In our first experiment, we analyse traces refinement expressions. For presentation purposes, we split our results into two tables. Table 7.1 presents the results of analysing the following triple-disjoint concurrent systems: a memory system (Mem), a client-server system (CS), a resource-allocation system (Res), and a routing-chain system (Rout and Rout2). We check a variety of properties using refinement expressions. These properties are informally described as follows. For Mem, we check the value read from address 0 is the last one written to it. For CS, we make sure that Client 0 alternates requests and responses. For Res, we ensure that User 0 acquires resources respecting their index order. In Rout, we check that a token rotates between components in a particular subsystem of this system, whereas in Rout2, we check that this subsystem behaves as a 1-place buffer.

	N	Approximate		Exact		
		ApprOx	**AppOxSMT**	FDR	FDRc	FDRp
Mem	50	0.26	0.11	*	0.31	0.16
	100	0.11	0.11	*	0.62	0.62
	200	0.16	0.16	*	1.32	*
	300	0.27	0.21	*	2.22	*
CS	10	0.21	0.97	*	+	0.31
	20	1.87	19.45	*	+	5.32
	30	11.34	114.99	*	+	21.81
	40	58.58	*	*	+	*
Res	40	0.36	0.37	*	1.02	30.46
	60	0.26	0.51	*	2.67	131.14
	80	0.37	0.92	*	6.28	*
	100	0.67	1.42	*	13.84	*
Rout	40	1.47	1.77	*	48.21	*
	60	3.12	3.57	*	219.89	*
	80	6.18	6.88	*	*	*
	100	12.54	12.94	*	*	*
Rout2	40	2.52	2.27	*	48.16	*
	60	4.78	4.12	*	216.68	*
	80	8.74	7.43	*	*	*
	100	15.65	13.84	*	*	*

Table 7.1: Results for traces refinement checking. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the refinement expression being tested. + means that no efficient compression technique could be found.

Table 7.2 presents the results of analysing the following triple-disjoint concurrent systems: system of lifts (Lift and Lift2), the asymmetric solution to the dining philosophers (Phil and Phil2), a non-fillable ring system (Ring), and a version of Milner's scheduler (Rot). We check the following properties for these systems. For Lift, we check that Lift 0 works by opening its doors in a floor and closing it at the same floor. For Lift2, we check that a Floor 0 works by letting a lift at a time open and close its doors when at this floor. For Phil, we test that at most $N/2$ philosophers can be eating at the same time, whereas in Phil2, we check that Philosophers 0 and 1 cannot be eating at the same time. For Ring, we verify that all nodes cannot be full at the same time. Finally, in Rot, we check that a token rotates amongst components in this system.

For these refinement expressions, unlike complete/exact methods, our frameworks

	N	Approximate		Exact		
		ApprOx	**AppOxSMT**	FDR	FDRc	FDRp
Lift	10	0.27	0.46	4.52	+	0.11
	20	0.46	7.58	*	+	*
	30	1.12	84.28	*	+	*
	40	2.62	*	*	+	*
Lift2	10	0.11	0.16	4.22	+	6.17
	20	0.32	0.82	*	+	*
	30	0.52	1.17	*	+	*
	40	0.77	1.72	*	+	*
Phil	10	-	0.21	*	0.21	*
	15	-	0.31	*	0.66	*
	20	-	2.17	*	6.78	*
	25	-	21.36	*	112.49	*
Phil2	20	0.11	0.12	*	0.31	*
	40	0.16	0.16	*	0.97	*
	60	0.17	0.21	*	2.42	185.94
	80	0.26	0.31	*	5.88	34.16
Ring	25	-	*	*	*	*
	50	-	7.98	*	*	*
	75	-	11.71	*	*	*
	100	-	33.28	*	*	*
Rot	20	0.31	7.63	40.08	14.74	0.11
	40	0.92	*	*	*	0.16
	60	10.53	*	*	*	0.26
	80	262.04	*	*	*	*

Table 7.2: Results for traces refinement checking. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the refinement expression being tested. + means that no efficient compression technique could be found.

scale reasonably well, which seems to indicate its ability to cope with the state-space explosion problem. The SMT version of our framework appears to be generally less scalable than its SAT counterpart. Moreover, we have an anomalous behaviour of the SMT solver for Ring when $N = 25$. The solver seems to get stuck in a part of the search space that is not essential in showing unsatisfiability for this instance's generated formula. Some changes in parameters can make this instance be solved in a matter of seconds. These changes, however, hinder the performance of the solver for most of the other examples.

7.2.5.2 Stable-failures refinement checking

In our second experiment, we analyse stable-failures refinement expressions. For presentation purposes, we split our results into two tables. Table 7.3 presents the results for the analysis of the same systems analysed in Table 7.1. We check the following properties for them. For Mem, we check that any value can be written to any address at any time. For CS, we make sure that a response is offered to Client 0, after it has issued a request. For Res, we ensure the system is deadlock free. In Rout, we check that token-passing events are offered in a way that allows the token to rotate amongst components in a particular subsystem of this system, whereas in Rout2, we check that this subsystem behaves, in terms of stable failures, as a 1-place buffer.

		Approximate		Exact		
	N	**ApprOx**	**AppOxSMT**	FDR	FDRc	FDRp
Mem	50	0.11	0.11	*	0.42	*
	100	0.16	0.16	*	1.02	*
	200	0.27	0.26	*	3.47	*
	300	0.52	0.36	*	7.63	*
CS	10	0.42	1.67	*	+	*
	20	7.03	33.18	*	+	*
	30	49.16	173.29	*	+	*
	40	203.24	*	*	+	*
Res	40	1.12	230.37	*	4.22	*
	60	4.07	*	*	13.79	*
	80	15.94	*	*	38.69	*
	100	52.12	*	*	94.39	*
Rout	40	2.47	3.57	*	48.21	*
	60	4.78	7.18	*	219.48	*
	80	8.79	12.89	*	*	*
	100	15.85	21.71	*	*	*
Rout2	40	2.52	4.02	*	48.51	*
	60	4.83	7.73	*	219.09	*
	80	8.69	14.14	*	*	*
	100	16.05	23.21	*	*	*

Table 7.3: Results for stable-failures refinement checking. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the refinement expression being tested. + means that no efficient compression technique could be found.

Table 7.4 presents the results of analysing the same triple-disjoint systems analysed

in Table 7.2 except that we analyse the butler solution to the dining philosophers (But) instead of the asymmetric solution (Phil and Phil2). We check the following properties for these systems. For Lift, if Lift 0 open its door at a floor it must be able to close it at the same floor. For Lift2, after opening the its door for a lift, Floor 0 must be able to close it. For But, we check that at least one philosopher must be waiting to sit at the table at all times. For Ring, we check deadlock freedom. Finally, in Rot, we ensure that token-passing events are offered in a way that a token rotates amongst components in this system.

		Approximate		Exact		
	N	**ApprOx**	**AppOxSMT**	FDR	FDRc	FDRp
Lift	10	0.52	2.42	8.08	+	43.08
	20	3.97	42.90	*	+	*
	30	12.44	*	*	+	*
	40	29.62	*	*	+	*
Lift2	10	0.21	0.77	6.58	+	35.97
	20	2.37	20.01	*	+	*
	30	5.02	57.17	*	+	*
	40	8.73	70.15	*	+	*
But	20	-	2.37	*	+	*
	30	-	12.29	*	+	*
	40	-	50.01	*	+	*
	50	-	157.98	*	+	*
Ring	100	0.62	53.77	*	0.77	*
	200	1.17	*	*	1.82	*
	300	2.32	*	*	3.37	*
	400	3.97	*	*	5.47	*
Rot	20	0.16	5.72	42.04	18.20	0.61
	40	1.47	*	*	*	52.99
	60	14.84	*	*	*	*
	80	271.65	*	*	*	*

Table 7.4: Results for stable-failures refinement checking. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to verify systems. * 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 the refinement expression being tested. + means that no efficient compression technique could be found.

Unlike complete/exact methods, our frameworks scale reasonably well for these examples. This indicates its ability to cope with the state-space explosion problem. We point out that the normalisation process for the stable-failures property of CS is rather demanding, which considerably contributes to the rather poor scalability in

checking this example. The lack of scalability of our SMT-based approach for some of our examples seems to suggest that a better SMT refinement-checking encoding could be achieved. This conclusion is particularly apparent for Ring. Checking deadlock freedom for this example via a static property is substantially more scalable than checking its refinement-based counterpart.

Our results seem to suggest that replacing exact reachability by our approximations creates an useful refinement-checking framework in the sense it can reasonably quickly check non-trivial refinement expressions. Note, for many of these examples, our approach appears to be the only fully automatic alternative that can tackle them in a reasonable amount of time.

7.3 Conclusion

This chapter extends our investigation of approximate frameworks to the verification of CSP's traces and stable-failures refinement expressions.

We propose a watchdog-based framework that, broadly speaking, translates checking a refinement assertion into the verification of a static property. We adapt a known watchdog construction [GMR+03] to take the best our of our reachability approximations. Building on our previous frameworks, we detect candidate violations using SAT and SMT solving. Our experiments' results seem to show that, by using approximations, we were able to harness the expected power of SAT/SMT-based exploration, which appears to considerably tame the state-space explosion problem. They also seem to suggest that a better encoding for our SMT-based approach could be devised. To the best of our knowledge, this framework is the first approximate approach to check CSP's traces and stable-failures refinement expressions.

In [POR12], a precise framework using SAT-based exploration to check traces refinement is proposed. It combines a watchdog approach with SAT-based *bounded model checking* and *k-induction* to precisely look for traces violations. For instances without violations, namely, for which traces refinement holds, this framework does not scale. According to the framework's authors, it is, indeed, considerably less efficient than using simple explicit state-space exploration. This inability also supports our claim that our approximations are a key factor in unlocking the power of this sort of symbolic exploration.

Our framework's efficiency comes, however, at the cost of imprecision. It can only show that some refinement expressions hold, and we are entirely unable to show that a refinement expression does *not* hold. Moreover, even though we are able to guarantee

a number of properties for some interesting systems, there are still some technical challenges in reaching a fully fledged approximate framework for refinement checking. A few interesting properties are still out of reach for our framework. For instance, generally showing that a concurrent system does not allow message overtaking is beyond the capability of our framework. We intend to overcome this challenge by adding other approximation techniques that would capture further invariants. Another challenge is to better incorporate the structure of a refinement expression in our framework. Currently, the specification is treated as just another component. To improve our framework, however, we believe that the specification should have a special role and, consequently, a more specialised analysis.

We do *not* anticipate the possibility of extending our framework to check CSP's failures-divergences refinement. The reason is that liveness/progress properties do not seem to fit well with the kind of reachability approximations we use, and analysing divergence is a necessary requirement for checking this type of refinement. To capture such properties, we would probably need to use again a watchdog-based approach but, in this case, we believe that the complexity of the required watchdog would severely hinder the sort of reachability analysis we propose. A few approximate frameworks have been proposed to check liveness properties [OPRW13, FOSC16] but they use rather different types of analysis.

Chapter 8

Conclusion

This thesis has proposed a variety of frameworks and techniques to verify concurrent systems using reachability over-approximations. We have created a number of techniques to approximate reachability that draws inspiration from some core ideas used in the analysis of concurrent systems. Moreover, we have investigated the frameworks that arise from replacing exact reachability by our approximations. These frameworks look for a candidate violation that passes some test/predicate approximating reachability. They intentionally sacrifice completeness to gain efficiency. Their use of reachability over-approximations means that they are sound – if no candidate violations are found, no violations exist – but incomplete: if a candidate violation is found, we do not know whether it is reachable. This incompleteness is acceptable as long as testing that a state lies in this approximation is generally quicker than checking exact reachability. So, unlike traditional precise methods to combat the state-space explosion, approximate frameworks tend to efficiently check most systems and properties, albeit, in some cases, imprecisely. Our techniques could be used in the system-development cycle in conjunction with techniques for *under-approximating* the system's state space. They could be part of an iterative development process where under-approximations would find bugs, leading to the re-designing of the system, up until the point our over-approximations prove that the desired property holds.

In Chapter 3, we looked at the first core idea we were inspired by: local analysis. Sometimes a property can be established by examining small parts of the system. In these cases, verification frameworks could immensely benefit from analysing these small parts as opposed to the entire system. Guided by this idea, we have proposed a notion of subsystem reachability; it estimates the states a system can reach by looking at the states some subsystem can reach. This notion provides a way to capture and implement local analysis. As picking which subsystems should be used to approximate the state space of a system might be a daunting task, we have proposed the notion

of k-reachability. It combines subsystem reachability for a set of (up-to-)k-sized subsystems. Thus, by picking a $k \geq 2$, one has a reachability approximation that can readily power a verification framework.

In the same chapter, we proposed Pair: a framework that uses 2-reachability, i.e. pairwise analysis, to check deadlock freedom. It looks for a Pair candidate deadlock, namely, a blocked state that passes our 2-reachability test. Our analysis of Pair has shown it can check deadlock freedom for some well-behaved resource-allocation and client-server systems. Moreover, we have demonstrated that Pair is provably more precise than local-analysis-based methods that check for the absence of cycles of ungranted requests. Pair can, in particular, handle some non-hereditary deadlock free systems, while cycle-based methods are completely ineffective in tackling such systems. We implemented the Pair framework in our DeadlOx tool. Our practical evaluation of this implementation suggests it is fairly scalable.

Pair is unable, however, to show deadlock freedom when it depends on some invariants of triples or larger combinations of components. Generally, a framework using k-reachability cannot prove properties that depend on some invariant of subsystems involving more than k components. To alleviate this problem, we have proposed the PairPicking strategy. It allows the user to hand-pick some subsystems (i.e. triple or larger combinations of components) that are analysed in order to improve the precision of Pair. Global system invariants, however, cannot be efficiently captured by any of these techniques. This shortcoming is inherent to the pure use of local analysis.

We have improved on local analysis by proposing some techniques to capture global system invariants. In Chapter 4, we looked at the second core idea that inspired this thesis: synchronisation analysis, and the way it combines component invariants to capture global system invariants. We have proposed techniques that use component-synchronisation-analysis frameworks to compute component-state invariants, i.e. summaries of the behaviour leading a component to one of its states. Then, these summaries are used to detect whether components can consistently cooperate to reach a system state; if they cannot, our techniques deem this system state unreachable. While one approach tries to check whether components can consistently agree on the number of interactions they need to perform to reach a state, the other tries to establish whether components can agree on the order in which they cooperate.

Building on Pair, this chapter also proposed PairStatic: a framework that combines our pairwise-analysis and these two global-analysis techniques to check deadlock freedom. We show that, as intended, this framework improves on Pair as it can

capture and use some global system invariants to prove deadlock freedom. We also show that PairStatic can show deadlock free systems that are beyond the capabilities of similar approximative frameworks combining local and global analysis techniques. We extended DeadlOx to implement our PairStatic approach. The practical evaluation of this implementation suggests that PairStatic is generally more precise than similar frameworks. Furthermore, it also hints at the versatility of our component-state invariants. They can capture a variety of common mechanisms used by systems to avoid blocked states. For instance, PairStatic can show deadlock freedom for some well-behaved systolic-array-like, non-fillable, counting-based and token-based systems.

The global-analysis techniques embedded in PairStatic, however, can only incidentally capture token-based mechanisms; they can only capture some conservative token mechanisms for which the routes that tokens take around the system are somewhat predictable. Nevertheless, since token mechanisms are so ubiquitous in the context of concurrent systems, we decided to investigate techniques to better capture these mechanisms.

In Chapter 5, we looked at the third core idea that inspired our work: token invariants. This chapter introduces two techniques to capture token invariants. These techniques detect whether a system implements a token structure, no matter how unpredictable token routes might be. So, they try to consistently assign to each component state the number of tokens the component might hold at that point. While one technique detects conservative token structures, where the interaction of components can cause tokens to be exchanged but not destroyed or created, the other detects existential structures, where tokens can be created and destroyed but not completely annihilated from the system. A token structure naturally leads to a token invariant that can be used as a reachability approximation. For instance, in the conservative case, we can count how many tokens are held by components in the initial state and, because tokens are conserved, this number must remain the same for every reachable state. Therefore, system states for which the sum of tokens does not add up to this initial value have to be unreachable.

In this chapter, we also extended Pair to accommodate these token-structures-detecting techniques. This combination gave rise to the PairToken framework. Unlike PairStatic, this framework can find token structures, if they exist, regardless of how unpredictable they are. We extended DeadlOx to implement PairToken. Unsurprisingly, our practical evaluation suggests that this framework fares considerably well in proving deadlock freedom for systems that implement token mechanisms. It could even detect some useful token structures for systems that do not obviously implement a token

mechanism. For instance, we could find token structures for some systolic-array-like systems that we had not anticipated we would. Our evaluation also showed that PairToken can show deadlock freedom for some systems implementing non-fillable, counting-based, and priority-based mechanisms amongst others. Generally, our analysis seems to suggest that while generally less precise than PairStatic, PairToken is more accurate in proving deadlock freedom for systems implementing mechanisms that can be naturally captured by token structures.

Table 8.1 lists (in an index-like format) all the reachability approximations that we propose in these chapters. We present the predicates, their respective SAT/SMT constraints, one-sentence descriptions, and whether these predicates and constraints require triple disjointness. Note that even though the techniques behind $reach_2$, $reach_S$, $reach_R$ and $reach_D$ (and the abstract versions of these three predicates) do not require triple disjointness, the implementation of Pair and PairStatic do require input systems to be triple disjoint; these implementations rely on the *Blocked* constraint which does require triple disjointness. Our approximations try to capture flexible invariants/abstractions that are implemented by common interaction mechanisms. This flexibility means that with a few types of invariants we can capture a variety of mechanisms. We do not intend to capture all mechanisms but a relevant class of commonly used ones; so our frameworks can verify systems implementing a combination of such mechanisms.

The success of our approximate frameworks for deadlock-freedom verification motivated us to investigate which other properties could be effectively checked by frameworks using approximations instead of exact reachability. Since our approximations are in no way moulded to the analysis of deadlocks, they can be soundly used to check other properties. Still, we have to make sure that the verification frameworks they give rise to are efficient and reasonably precise.

In Chapter 6, we have investigated the use of our approximations in checking static properties. A static property is described in terms of the immediate behaviour of the system; a violating system state is specified by a combination of events it offers/refuses. These properties ensure no such violating state exists. They naturally fit into the kind of framework we propose because detecting violations can be simply encoded into a constraint on individual system states. We propose two types of static properties: global ones, which are analysed based on the overall behaviour of a system, and local ones, which are analysed based on the behaviour of subsystems of a system. Note that checking local properties involves the analysis of all (exponentially many) subsystems of a system. So, traditional verification frameworks are utterly ineffective

Predicate	Constraint	Description	TD
$reach_2$ (Def.3.7, p.57)	$Reach_2$ (Sec.3.3.3, p.70)	Local-analysis technique that combine pairwise reachability analyses for the components of systems.	No.
$reach_S$ (Def.4.5, p.94)	$Reach_S$ (Sec.4.3.2, p.118)	Synchronisation-analysis technique that uses suffixes to check if components can agree on an order in which they participate on system rules.	No.
$reach_R$ (Def.4.10, p.100)	$Reach_R$ (Sec.4.3.2, p.118)	Synchronisation-analysis technique that analyses whether components can cooperate to reach a state by using relations between the number of times that components participate on system rules.	No.
$reach_D$ (Def.4.13, p.102)	$Reach_D$ (Sec.4.3.2, p.118)	Synchronisation-analysis technique which uses suffixes to check if components can agree on an order in which they participate on system rules.	No.
$reach_S^{DA}$ (Sec.4.2.3.1, p.105)	$Reach_S^{DA}$ (Sec.4.3.2, p.118)	The technique of $reach_S$ combined with data abstraction.	No.
$reach_R^{DA}$ (Sec.4.2.3.1, p.105)	$Reach_R^{DA}$ (Sec.4.3.2, p.118)	The technique of $reach_R$ combined with data abstraction.	No.
$reach_D^{DA}$ (Sec.4.2.3.1, p.105)	$Reach_D^{DA}$ (Sec.4.3.2, p.118)	The technique of $reach_D$ combined with data abstraction.	No.
$reach_D^{CA}$ (Sec.4.2.3.2, p.108)	$Reach_D^{CA}$ (Sec.4.3.2, p.118)	The technique of $reach_D$ combined with component-specific abstraction.	No.
$reach_{C_{\mathbb{B}}}$ (Def.5.4, p.136)	$Reach_{C_{\mathbb{B}}}$ (Sec.5.3.2, p.156)	Token-detection technique that generates a conservative token invariant.	Yes.
$reach_{E_{\mathbb{B}}}$ (Def.5.6, p.139)	$Reach_{E_{\mathbb{B}}}$ (Sec.5.3.2, p.156)	Token-detection technique that generates an existential token invariant.	Yes.
$reach_C^{ce}$ (Def.5.9, p.143)	$Reach_{C_{\mathbb{B}}}^{ce}$ (Sec.5.3.2, p.156)	Counter-example-guided token-detection technique that generates a conservative token invariant.	Yes.
$reach_{E_{\mathbb{B}}}^{ce'}$ (Def.5.13, p.145)	$Reach_{E_{\mathbb{B}}}^{ce'}$ (Sec.5.3.2, p.156)	Counter-example-guided token-detection technique that generates an existential token invariant.	Yes.

Table 8.1: Summary of approximations and corresponding SAT/SMT constraints. The TD column depicts whether the predicate/constraint requires triple disjointness.

in handling such properties. Finally, we point out that many common properties can be naturally framed as static properties. For instance, ensuring deadlock freedom, mutual exclusion, or, generally, that some marked error state cannot be reached are common examples of static properties.

In this chapter, we have also proposed StaticProperty: a framework that combines our approximations to check static properties. We demonstrate that this framework can verify such properties for non-trivial systems. This chapter introduces a few examples of static properties that can be checked thanks to the ability of our approximations in capturing, among others, counting-based, token-based, resource-allocation-based system mechanisms. We implemented this framework in our ApprOx tool. Our practical evaluation suggests that many state-based properties can be conveniently formulated using our notation and effectively checked by ApprOx. This evaluation also attests the versatility of our approximations in capturing different sorts of mechanisms implemented by systems to avoid bad states. It also seems to show that, despite requiring the analysis of exponentially many subsystems, local static properties can be efficiently tackled thanks to the power of solvers in handling the sort of existential quantification these properties are based upon.

Static properties naturally capture state-based properties but they are unable to simply capture path-based behavioural properties. They fail to simply capture, for instance, that a state/event must happen after some other state/event. To overcome this issue, we decided to investigate whether/how our approximations could be used for refinement checking.

In Chapter 7, we proposed RefinementChecking: a framework that combines our approximations to check CSP's traces and failures refinement expressions. This framework adapts a watchdog-based approach to turn checking a refinement expression into verifying an equivalent static property. This chapter has also illustrated how the combination of our approximations can capture system invariants that are strong enough to prove some non-trivial refinement expressions. Also, we extended ApprOx to implement this framework. This implementation's practical evaluation attests the flexibility of our invariants in capturing a variety of mechanisms commonly implemented by concurrent systems to avoid undesired states. Our analysis seems to suggest that our pairwise-analysis approximation plays a particularly important role in tightly capturing these path-based properties.

These frameworks and techniques do not represent by any means a definite solution to the state-space explosion problem. Nevertheless, they do represent a new set of robust tools to be used against it. Traditional imprecise frameworks normally

approximate a property by some condition that can be checked in polynomial time. Our frameworks, however, are based around *NP*-hard candidate-detection problems. This result can be interpreted as highlighting the fact that we tackle a different class of systems and properties if compared to traditional approximate frameworks. Our approaches seem to be generally more precise than traditional approximate methods, while faring similarly in terms of verification time. So, we believe they provide a better compromise between speed and precision. Also, since they are consistently faster than exact verification methods, they could be used as a preliminary step in showing that a system preserves some desired property. To the best of our knowledge, in the setting of message-passing concurrent systems, our work seems to be the only one that use approximations to verify such a general class of properties.

The *NP*-hardness of these problems implies that there is no known polynomial-time algorithm that can power our verification frameworks. That said, we have built upon the efficiency of modern SAT and SMT solvers in tackling this kind of problems to create effective implementations for our frameworks. DeadlOx and ApprOx seem to benefit from the synergy between the compact encodings we propose and the heuristic search implemented by these solvers. This claim is corroborated by the results that we obtain from evaluating them in practice. Many conditions that are not known to be generally checkable in polynomial time can be efficiently decided by our implementations. This perspective guided and motivated the work presented in this thesis, and we hope that the success of our work encourages further research into other relevant classes of approximations that are not necessarily known to be polynomially checkable but that can be efficiently checked by these solvers.

Table 8.2 lists (in an index-like fashion) all the computational-complexity results that we prove in this thesis. It presents results for the problems underlying all the verification frameworks investigated in this thesis. Note that we have not attempted to prove any results about the exact problem of checking local and global static properties, this is left as future work.

Another interesting point can be made about our work. There have been some ineffective attempts at applying symbolic exploration in the analyse of concurrent systems. Most of them did not manage to unleash the same checking power that SAT/SMT solvers bring to other domains, such as hardware and software verification. In this light, our work can be understood as demonstrating that approximations are a means to harness all the verification power these solvers can offer in the context of analysing concurrent systems. Thus, the work in this thesis should also encourage

Deadlock freedom	
Framework	Complexity result
Exact	*PSPACE*-complete (Thm.2.16, p.27)
Pair	*NP*-complete (Thm.3.18, p.67)
PairStatic$_R$	*NP*-complete (Thm.4.24, p.118)
PairStatic$_D$	*NP*-hard (Thm.4.24, p.118)
PairToken$_{\mathbb{B}}$	*NP*-complete (Thm.5.17, p.156)
PairToken$_{ceg}$	*NP*-complete (Thm.5.17, p.156)

Local static property	
Framework	Complexity result
Exact	?
StaticProperty$_{SAT}$	*NP*-hard (Thm.6.13, p.181)
StaticProperty$_{SMT}$	*NP*-hard (Thm.6.13, p.181)

Global static property	
Framework	Complexity result
Exact	?
StaticProperty$_{SAT}$	*NP*-complete (Thm.6.13, p.181)
StaticProperty$_{SMT}$	*NP*-hard (Thm.6.13, p.181)

Traces refinement	
Framework	Complexity result
Exact	*PSPACE*-complete (Thm.2.19, p.33)
RefinementChecking$_{SAT}$	*NP*-complete (Thm.7.11, p.208)
RefinementChecking$_{SMT}$	*NP*-hard (Thm.7.11, p.208)

Stable-failures refinement	
Framework	Complexity result
Exact	*PSPACE*-complete (Thm.2.22, p.35)
RefinementChecking$_{SAT}$	*NP*-complete (Thm.7.11, p.208)
RefinementChecking$_{SMT}$	*NP*-hard (Thm.7.11, p.208)

Table 8.2: Summary of computational complexity results introduced in this thesis. A question mark indicates that we do not know (i.e. have not proved) the complexity of the corresponding problem.

further investigation into understanding and proposing new ways to harness the verification power of these solvers.

Finally, we reinforce that the ideas in this thesis are not CSP-specific and should transfer easily to any formalism where systems are (can be) described as pairwise-interacting LTSs. DeadlOx/ApprOx uses FDR4 to obtain supercombinator machines from systems described using CSP, but analogous tools could be created for other notations by replacing its use of FDR4 to generate such machines.

8.1 Limitations and future work

We have assumed throughout this work that systems are triple-disjoint. Although many systems naturally fit into this category, many others do not. So, it would be interesting to lift this requirement to investigate how the sort of approximate framework we propose fares in analysing non-triple-disjoint systems. It would be particularly interesting to evaluate whether precision is hindered by this extension. Many of our definitions can readily tackle non-triple-disjoint systems but some others need adapting. Notably, our token-detection techniques need to be significantly adapted to handle such systems.

In this work, we have informally hinted at many classes of systems and properties that can be successfully tackled by our frameworks. Formally systematising these (and other) classes would help to guide system designers in building correct systems that can be efficiently checked by our tools. Along the lines of the work in [Mar96], we could formalise the system mechanisms that can be captured by our approximations. Given these precise descriptions, we could then investigate which common static properties and refinement expressions can be successfully tackled by our frameworks.

In this work, we only tangentially investigated how to best encode our candidate-violation-detection problems as SAT and SMT problems and also how to best tune SAT and SMT solvers to tackle them. On the SAT side, we have investigated some different encodings and parameters to tune our implementation. The use of sorting networks in Chapter 4 and the configuration of variables polarity in Chapter 5 resulted from this investigation. On the SMT side, however, we merely relied on linear integer arithmetic, which is commonly recognised as a theory that is efficiently handled by SMT solvers, to encode our constraints and no solver tuning was attempted. The problems of finding good encodings and tuning the solver are very complex in the SMT domain. To begin with, there are many (combinations of) theories to try out each of which might lead to different encodings of the same problem. On top of that,

a SMT solver is in itself a combination of many solvers each of which relies on its own parameters and peculiarities. So, there are endless possible combinations of encodings and configurations to evaluate. This future investigation is particularly motivated by the lack of scalability for some of our encodings. For instance, the slow convergence of our minimisation procedures in detecting token structures for our SMT encoding in Chapter 5 suggests that there may be a better way to carry this out. One way to improve it would be to cast this minimisation problem as an optimisation problem, which can be tackle by some SMT solvers, as opposed to our iterative procedure. This lack of scalability is also evidenced in our SMT encoding for RefinementChecking in Chapter 7. Thus, further work on tuning our SAT and, particularly, our SMT encodings should be encouraged.

A rather interesting and relevant point that needs investigation is the connection between our original problems and their SAT/SMT counterparts, and in particular the interaction between the structure of our SAT/SMT encodings and the heuristics used by these solvers. This investigation could offer some fundamental explanation/evidence as to why our problems are efficiently tackled by these solvers. This investigation could even lead to the creation of a solver specifically designed to tackle the kind of candidate-violation-detection problem we handle.

Our approximations rely on flexible invariants that can tackle many sorts of mechanisms implemented by concurrent systems to avoid undesired states. However, unsurprisingly, there are still mechanisms that cannot be captured by our frameworks. For instance, generally capturing that a system does not allow message overtaking is beyond the capability of approximations[1]. To overcome such issues related to the inability of our frameworks to capture some mechanism, one can propose new approximation techniques and tests to finely capture the desired invariant. It should be pointed out that some practical evaluation must be made as to ensure this addition does not hinder the efficiency (and, consequently, the entire purpose of giving up completeness) of the resulting verification framework.

We anticipate that the sort of approximation we use should be fairly ineffective in tackling progress properties such as proving divergence freedom or that some event is eventually performed. We believe that the watchdog-like approach required to translate such a property to a static-like property would be so complex that our approximations would not be able to properly analyse this watchdog-based system.

[1]We focused in outlining the systems that our frameworks can tackle and not in investigating which systems they cannot handle. So, we have only particularly identified this message-overtaking invariant as a interesting property that cannot be captured by our arsenal of techniques. That said, there should be other interesting properties our frameworks are unable to capture.

There might be, however, less obvious ways by which such a translation can be achieved or by which our approximations could be adapted directly to test such properties. If such an approach is devised, our approximations could give rise to an useful verification framework for these properties.

Lastly, we discuss the possibility of investigating under-approximation techniques. This entire work revolves around techniques for over-approximating reachability that can power efficient verification frameworks. These frameworks can show the validity of a property but they are unable to prove that a property has been violated; as discussed, this incompleteness is the cornerstone of such frameworks. So, they do not provide undeniable counter-examples witnessing such a violation. Our frameworks merely present a candidate that might hint at a real violation but it might as well just be an unreachable state. Hence, an interesting research direction is to investigate whether the ideas behind our techniques could be used in some form to under-approximate the state space of a system. These techniques could give rise to approximate frameworks that could quickly detect true counter-examples.

This thesis investigates the complexity of (verification-related) problems that take as an input a supercombinator machine instead of a sequential (single-component) computational device. We revisit some problems that have been long studied in the context of single-component machine and prove new results in the context of supercombinator machines. It would be interesting to further study/investigate the class of decision problems that analyse supercombinator machines and prove further complexity results about them. For instance, we have not investigated the complexity of exactly deciding whether a given static property (local or global) holds.

Bibliography

[ABB+13] Paul C. Attie, Saddek Bensalem, Marius Bozga, Mohamad Jaber, Joseph Sifakis, and Fadi A. Zaraket. An Abstract Framework for Deadlock Prevention in BIP. In *FORTE*, number 7892 in LNCS, pages 161–177. Springer, 2013.

[ABB+18] Paul C. Attie, Saddek Bensalem, Marius Bozga, Mohamad Jaber, Joseph Sifakis, and Fadi A. Zaraket. Global and local deadlock freedom in BIP. *ACM Trans. Softw. Eng. Methodol.*, 26(3):9:1–9:48, 2018.

[AC05] Paul C. Attie and Hana Chockler. Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In *VMCAI*, pages 465–481. Springer, 2005.

[AFDR80] Krzysztof R. Apt, Nissim Francez, and Willem P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):359–385, 1980.

[AGRR16a] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Efficient deadlock-freedom checking using local analysis and SAT solving. In *IFM*, number 9681 in LNCS, pages 345–360. Springer, 2016.

[AGRR16b] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Tighter reachability criteria for deadlock freedom analysis. In *FM*, number 9995 in LNCS. Springer, 2016.

[AGRR17a] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. The automatic detection of token structures and invariants using SAT checking. In *TACAS*, number 10206 in LNCS, pages 249–265. Springer, 2017.

[AGRR17b] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Checking static properties using conservative SAT approximations for reachability. LNCS, 2017.

[AGRR18] Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Experiment package, 2018. Available at: `www.cs.ox.ac.uk/people/pedro.antonino/thepkg.zip`.

[AOS+14] Pedro Antonino, Marcel Medeiros Oliveira, Augusto Sampaio, Klaus Kristensen, and Jeremy Bryans. Leadership election: An industrial SoS application of compositional deadlock verification. In *NFM*, volume 8430 of *LNCS*, pages 31–45, 2014.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, March 1983.

[AS09] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009.

[ASW14] Pedro Antonino, Augusto Sampaio, and Jim Woodcock. A refinement based strategy for local deadlock analysis of networks of CSP processes. In *FM*, volume 8442 of *LNCS*, pages 62–77, 2014.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BBH+09] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[BBL+16] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Software and System Modeling*, 15(2):427–451, 2016.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI 2003*, PLDI '03, pages 196–207. ACM, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BGL+11] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2: Towards efficient correctness of incremental design. In *NFM*, pages 453–458, 2011.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic Generation of Invariants. *Form. Methods Syst. Des.*, 15(1):75–92, July 1999.

[BR91] Stephen D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, **4**:209–230, 1991.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[CC77] Patrick Cousot and Radhia Cousot. 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*, POPL '77, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN*

Symposium on Principles of Programming Languages, POPL '79, pages 269–282, 1979.

[CCO+05] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[CDP06] Gerardo Canfora and Massimiliano Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, pages 78–105. Springer, 2006.

[CES71] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, **3**(**2**):67–78, 1971.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CGRX06] Sadie Creese, Michael Goldsmith, Bill Roscoe, and Ming Xiao. Bootstrapping multi-party ad-hoc security. In *SAC*, pages 369–375, 2006.

[CK94] Shing Chi Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug 1994.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. April 1995.

[CR99] S. J. Creese and A. W. Roscoe. Formal verification of arbitrary network topologies. In *PDPTA*, pages 1033–1039, 1999.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[Dat89] Naiem Dathi. *Deadlock and Deadlock Freedom.* DPhil thesis, University of Oxford, 1989.

[DCCN04] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, October 2004.

[Dij68] Edsger W. Dijkstra. The structure of the “THE”-multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968.

[DKW08] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[Eng] Roberto Baldwin, The world now has a smart toaster. Available: `https://www.engadget.com/2017/01/04/griffin-connects-your-toast-to-your-phone/` [Last accessed: 19 December 2017].

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[FF10] D. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, NJ, USA, 2010.

[FOSC16] Madiel S. Conserva Filho, Marcel Vinicius Medeiros Oliveira, Augusto Sampaio, and Ana Cavalcanti. Local livelock analysis of component-based models. In *ICFEM*, pages 279–295, 2016.

[GJR14] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. Sat modulo graphs: Acyclicity. *JELIA 2014*, pages 137–151, 2014.

[GMR+03] Michael Goldsmith, Nick Moffat, Bill Roscoe, Tim Whitworth, and Irfan Zakiuddin. Watchdog transformations for property-oriented model-checking. In *FME*, pages 600–616, 2003.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In *TACAS*, volume 8413 of *LNCS*, pages 187–201, 2014.

[GRHRW15] Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NFM*, volume 9058 of *LNCS*, pages 188–203. Springer, 2015.

[GS10] Stefan Gruner and T. J. Steyn. Deadlock-freeness of hexagonal systolic arrays. *Inf. Process. Lett.*, 110(14-15):539–543, 2010.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD*, **2**(**2**):149–164, 1993.

[HKPM97] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq proof assistant: a tutorial: version 6.1.* PhD thesis, Inria, 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hor] Nachum Dershowitz, Software horror stories. Available: `https://www.cs.tau.ac.il/~nachumd/horror.html` [Last accessed: 19 December 2017].

[JL16] Loig Jezequel and Didier Lime. Lazy Reachability Analysis in Distributed Systems. In Josée Desharnais and Radha Jagadeesan, editors, *CONCUR 2016*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[KS90] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43 – 68, 1990.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[LB05] Michael Leuschel and Michael Butler. Combining CSP and B for specification and property verification. In *FM'2005*, volume 3582 of *LNCS*, pages 221–236. Springer-Verlag, January 2005.

[LMC11] Christian Lambertz and Mila Majster-Cederbaum. Analyzing Component-Based Systems on the Basis of Architectural Constraints. In *FSEN*, pages 64–79. Springer, April 2011.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *TACAS '96*, pages 147–166, 1996.

[LS04] Shuvendu K Lahiri and Sanjit A Seshia. The UCLID decision procedure. In *CAV*, volume 3114, pages 475–478. Springer, 2004.

[Mar96] Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[MJ97] J.M.R. Martin and S.A. Jassim. An efficient technique for deadlock analysis of large scale process networks. In *FME '97*, pages 418–441, 1997.

[MS01] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Sci. Comput. Program.*, 40(1):59–96, 2001.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OAR+16] M. V. M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, and A. W. Roscoe. Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing*, pages 1–68, 2016.

[OCS17] Rodrigo Otoni, Ana Cavalcanti, and Augusto Sampaio. 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*, pages 107–124, 2017.

[OPRW13] Joël Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. A static analysis framework for livelock freedom in CSP. *LMCS*, **9(3)**, 2013.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[Pet77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Dept, Aarhus University, 1981.

[POR12] Hristina Palikareva, Joël Ouaknine, and A.W. Roscoe. SAT-solving in CSP trace refinement. *Science of Computer Programming*, 77(10):1178 – 1197, 2012.

[Ram11] Rodrigo Teixeira Ramos. *Systematic Development of Trustworthy Component-based Systems*. PhD thesis, Universidade Federal de Pernambuco, 2011.

[Ray89] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.

[RD87] A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Inf. Comput.*, **75(3)**:289–327, 1987.

[RGG+95] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression

for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS*, pages 133–152, 1995.

[Ros87] A. W. Roscoe. Routing messages through networks: an exercise in deadlock avoidance. In Muntean et al., editor, *Programming of Transputer Based Machines: Proceedings of 7th occam User Group Technical Meeting*, Amsterdam, 1987. IOS B.V.

[Ros98] A. W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[RS90] John H. Reif and Scott A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, Feb 1990.

[RSM09] Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. Systematic development of trustworthy component systems. In *FM*, pages 140–156, 2009.

[RW06] A. W. Roscoe and Zhenzhong Wu. Verifying statemate statecharts using csp and fdr. In *ICFEM*, pages 324–341, 2006.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[SD82] C. S. Scholten and Edsger W. Dijkstra. *A Class of Simple Communication Patterns*, pages 334–337. Springer New York, New York, NY, 1982.

[Sip12] M. Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2012.

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *CAV*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.

[SNM09] Augusto Sampaio, Sidney Nogueira, and Alexandre Mota. Compositional verification of input-output conformance via CSP refinement checking. In *ICFEM*, pages 20–48, 2009.

[Tar95] Gaston Tarry. Le probleme des labyrinthes. *Nouvelles annales de mathématiques, journal des candidats aux écoles polytechnique et normale*, 14:187–190, 1895.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Tse68] G Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[Wir] Simson Garfunkel, History's worst software bugs, Wired, 08 November 2005. Available: `https://www.wired.com/2005/11/historys-worst-software-bugs/` [Last accessed: 02 December 2017].

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[YY91] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 49–59. ACM, 1991.