Efficient verification of sequential and concurrent systems

Stefan Schwoon

▶ To cite this version:

Stefan Schwoon. Efficient verification of sequential and concurrent systems. Formal Languages and Automata Theory [cs.FL]. École normale supérieure de Cachan - ENS Cachan, 2013. tel-00927066

HAL Id: tel-00927066
https://theses.hal.science/tel-00927066

Submitted on 10 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient verification of sequential and concurrent systems

Mémoire d'Habilitation à Diriger des Recherches

Stefan Schwoon

December 2013

This thesis is to be defended on 6 December 2013 before a committee composed of

- Eike Best (reviewer)
- Javier Esparza
- Stefan Haar
- Petr Jančar (reviewer)
- Claude Jard
- Salvatore La Torre
- Helmut Seidl (reviewer)

This document reports on joint work with

Paolo Baldan
Ahmed Bouajjani
Alessandro Bruni
Barbara König
Andrea Corradini
Javier Esparza
Stefan Haar
Somesh Jha
Pradeep Kanade
Christian Kern
Victor Khomenko
Stefan Kiefer
Morten Kühnrich
David Melski
Thomas Reps
César Rodríguez
Jiří Srba
Jan Strejček
Stuart Stubblebine
Dejvuth Suwimonteerabuth
Hu Wang

Contents

1 Introduction **7**

2 Pushdown systems **15**

2.1 Basics about pushdown systems 16
2.1.1 Pushdown systems and reachability 17
2.1.2 Related constructions 23
2.1.3 Modelling programs as PDS 24
2.2 Weighted PDS 26
2.2.1 Bounded semirings 29
2.2.2 Unbounded semirings 37
2.2.3 Applications 41
2.3 Other PDS-related contributions 44
2.3.1 Tool development 45
2.3.2 Verification of concurrent PDS 45
2.3.3 Abstraction-refinement for symbolic PDS 48
2.3.4 Authorization systems 51

3 Petri nets **55**

3.1 Basics of c-nets and unfoldings 58
3.1.1 Contextual nets 58
3.1.2 Encodings 60
3.1.3 Occurrence nets, unfoldings, and prefixes 61
3.2 Making unfoldings more efficient 63
3.2.1 An abstract algorithm 65
3.2.2 Efficient construction of c-net unfoldings 69
3.2.3 Verification of c-net unfoldings 75
3.2.4 Contextual merged processes 79
3.3 Other contributions 83

3.3.1 The reveals relation 84
3.3.2 Weak diagnosis 87
3.3.3 A note on depth-first-search order 88
3.3.4 Improved detection of possible extensions 90

4 Conclusion **93**

Chapter 1

Introduction

When the author of this document was still a child, electronic computers had been in existence for thirty years but an average person would not personally have seen or used one. Since then, we have witnessed their gradual but incessant invasion into our lives, to the point of becoming omnipresent: laptops, smartphones, embedded systems in cars, etc.

With this development comes an undeniable need for ensuring that computer systems function correctly, and various methodologies have been developed for this purpose. *Software-engineering techniques* aim to reduce bugs in practice through a rigorous production process; or to detect them through systematic testing, which can find bugs but not guarantee their absence. By contrast, *formal methods* provide means for rigorously specifying the desired behaviour of a system, making a precise model of its actual behaviour, and then verifying whether that actual behaviour corresponds to the specification. Thus, one can prove either the presence or the absence of bugs.

Of particular interest is the case where the verification step happens automatically, i.e. is carried out by a computer. That is a challenging problem, for reasons that will be explained in a moment. Nonetheless, large progress has been made towards this end during the last thirty years. This is reflected by the fact that companies like Intel or Microsoft today employ large teams of verification specialists.

The two principal scientific challenges to automated verification are the decidability boundary and the effect of state-space explosion.

As for the first, it is well-known that all non-trivial behavioural properties of computer programs are *undecidable* in general, meaning that it is provably impossible to write a computer program A that analyzes another computer

program B and tells the user whether B can reach a certain program state, terminate, or satisfy other non-trivial specifications concerning its behaviour. More precisely, this is the case when when B can be an arbitrary program that has all the capabilities of of an ordinary ("Turing-powerful") computer. However, when the features of B are restricted, automated verification may still be possible. It is easy when B can only attain finitely many different states – it suffices to enumerate them all. Then again, interesting system often attain infinitely many states, for various reasons: they may handle data with an infinite domain (e.g., integers), handle an unbounded amount of data (lists), have infinite control structures (recursion), consist of an arbitrary number of components or participants (e.g., an network protocol), etc.

Even when the number of possible states is finite, that number may still be very large. Indeed, a simple program that manipulates a 32-bit integer variable and nothing else, can a priori have 2^{32} different states. The term *state-space explosion* describes the observation that even seemingly simple systems may have a huge number of different states, rendering naïve methods for automatic verification infeasible. Apart from data, another common source of state-space explosion is concurrency, due to the different orders in which concurrent threads can be executed.

Thus, there are two major endeavours in formal-verification research: (i) to find cases where automated verification is still possible, and (ii) to make the techniques efficient for large state-spaces.

As for (i), the idea is to identify the features and specifications, or combinations thereof, that still allow verification, and develop algorithms for that purpose. Failing this, another approach is to derive approximation techniques for certain subclasses, i.e. algorithms that err only on one side: an algorithm that over-approximates the reachable state space is guaranteed to find all existing errors but may report additional spurious ones, whereas an under-approximating algorithm guarantees correctness only for executions that do not exceed certain bounds.

As for (ii), one tries to identify 'smart' techniques that alleviate the effects of state-space explosion. Some general ideas in this domain are *symbolic verification*, where adequate data structures are used to represent many states concisely and manipulate them all at once; *reduction*, where the verification problem is reduced to a smaller, equivalent one; or *abstraction*, where part of the information about B is deliberately omitted, leading to a smaller, non-equivalent problem whose solution errs only in one direction.

This results reported in this thesis contribute to these realms. My main

interest has been on algorithmic aspects. This is motivated by the observation that asymptotic worst-case complexity, often used to characterize the difficulty of algorithmic problems, is only loosely related to the difficulty encountered in solving those problems in practice. Indeed, a good choice of algorithms and careful implementation can make all the difference between rendering a problem feasible or infeasible in practice. Well-known examples include, for instance, SAT checkers to solve NP-complete problems, or binary decision diagrams for symbolic verification. For this reason, many papers on which the thesis is built are accomodated by implementations and experimental results.

Content of this document

This document represents a synthesis of selected parts of my research, carried out since my PhD. This presentation focuses on the two most significant aspects of my work, each corresponding to one chapter. The two main formalisms used in this paper are pushdown systems and Petri nets. Both are fundamental notions of computation, and both offer, in my opinion, particularly nice opportunities for combining theory and algorithmics. In both chapters, I chose to highlight one particular line of research in more detail and give extended abstracts of the other contributions.

In all these works I have profited greatly from the collaboration with other colleagues, to whom I extend my heartfelt thanks. They are duly credited in the beginning of each section in Chapters 2 to 3.

Chapter 2 is dedicated to verification methods for pushdown systems (PDS). These are finite automata equipped with a stack; since the height of the stack is not bounded, they represent a class of infinite-state systems that model programs with (recursive) procedure calls. Fortunately, this interesting class of systems admits automated verification. Already for my PhD thesis [Sch02a], I investigated model-checking algorithms for PDS and a tool called Moped implementing these algorithms.

The main line of work presented in this chapter is a framework for *weighted PDS*. This was the first research interest I pursued after my PhD and therefore had more time than the others to make an impact. It also provided the basis for the other contributions in this chapter. In a weighted PDS, a transition rule is equipped with a weight from an idempotent semiring, where the weight may express quantitative aspects of the action represented by the

rule. This allows, for instance, to express general data-flow problems and has applications in program analysis and verification. In the work originally presented in [RSJ03, RSJM05], those semirings had to be *bounded*, i.e. without any infinite descending chains among the weights. Later, the setting was extended to allow infinite descending chains [KSSK09a].

Various other contributions in this area will be presented more briefly. The first among these are extensions to concurrent PDS. Verification methods for concurrent PDS are challenging not only due to state-space explosion, but also because of the decidability boundary. The contributions in this area are based on the *context-bounded* approach, where one only considers executions in which the active process changes no more than a fixed number of times. We first extended the approach and made it more expressive [BESS05]. The applicability of the context-bounded approach in its original form was hampered by the difficulty to combine it with symbolic techniques. In [SES08], we tackled this problem and developed a variant suitable for BDD-based methods.

Another contribution in this category was a framework for integrating counterexample-based abstraction refinement into BDD-based pushdown verification [EKS06, EKS08b]. I developed some tools in this area: The Wpds library [Schc] is based on the eponymous formal framework and served as the basis for a new implementation of Moped. The latter matured into a user-friendly tool for analyzing Java programs (jMoped) through the work of Dejvuth Suwimonteerabuth, a PhD student in our group. jMoped also serves as a test environment for Java programs [SSE05, SBSE07].

Specifying authorizations is another, particularly interesting application of pushdown systems. Indeed, PDS serve as a suitable model for expression authorization grants. Such a model has been proposed under the name SPKI/SDSI standard (RFC 2693). Therefore, algorithms for pushdown reachability provide an implementation of access-control policies. In this chapter, we study several theoretical and practical aspects of this relationship.

First, weighted PDS can give richer semantics to SPKI/SDSI, which are relevant in a security-related context, such as privacy, recency, validity, and trust [SJRS03]. Second, alternating PDS naturally enable intersection of certificates, a possibility provided for by the RFC, but, due to its apparent complexity, ignored by all previous algorithms for SPKI/SDSI. While reachability for alternating PDS is EXPTIME-complete in general, we show that a sensible restriction suitable for this application yields a polynomial subcase [SSE06]. As a final extension of the formalism, probabilistic PDS are shown

to be a suitable tool for computing the reputation of individuals within a group [BESS08].

On the practical side, we showed how to implement SPKI/SDSI authorization in a distributed setting [JSWR06] and how to integrate it into Kerberos [WJR+06]. I recently had the opportunity to put this work into practice in the server used by the computer science department of ENS Cachan to manage, e.g., courses, students, their marks, etc.

Chapter 3 details the work done on Petri nets. Unlike PDS, which are primarily suited to express sequential systems, Petri nets model concurrent systems. My contributions in this area all concern *unfoldings*. In a nutshell, the unfolding of a net N is an acyclic version of N in which loops have been unrolled.

Again, the chapter focuses on one line of work, this time a more recent one, which went towards improving the conciseness in cases that ordinary unfoldings cannot handle well: (i) concurrent read access and (ii) choice. For (i), we investigated the extension of Petri nets with read arcs. This part includes the development of the necessary theoretical background, [BCKS07, BCKS08, BBC+10], algorithms and data structures for efficient construction of such unfoldings [RSB11, BBC+12], their verification [RS12], and finally, taking into account aspect (ii), their combination with the notion of merged processes [RSK13].

Other contributions are described more concisely. The first among these concerns the use of unfoldings in partially observable concurrent systems. Here, we investigated the use – and efficient computation of – a so-called *reveals* relation that allows to determine the occurrence of one event a from the observation of another event b in a concurrent system, even if a and b are neither causally nor temporally related [HKS11, HKS13]. We then proposed an improved diagnosis method for concurrent systems based on Petri net unfoldings [HRS13].

A minor contribution consisted in showing the impossibility of correctly constructing finite unfolding prefixes using depth-first search [EKS08a]. Another was an efficiency improvement for the construction of net unfoldings [RS13].

Meanwhile, I developed the tool Mole [Schb], a fast unfolder for Petri nets. An equivalent tool for nets with read arcs, called Cunf [Rod], was developed by PhD student César Rodríguez during our work on the subject.

Other contributions not in this thesis

Some other contributions are not represented in the following chapters; they are briefly listed below:

- Together with Javier Esparza and Andreas Gaiser, I investigated algorithms for checking emptiness of Büchi automata, a central problem in verification. In the automata-theoretic approach to model-checking linear-time temporal logic [VW86], a temporal specification is violated if and only if a certain Büchi automaton has an accepting run. We investigated algorithms for explicit-state model checking where said automaton is constructed on the fly, of the type popularized by the tool Spin [Hol04]. We proposed improvements of existing algorithms that decrease their memory requirements and/or their response time in case of an accepting run exists, including some algorithms that are optimal in this respect. The algorithms were investigated from a theoretical [SE05] and a practical perspective [GS09].

- While I was on leave at Microsoft Research, we investigated a systematic approach to discover security holes in the configuration of operating systems such as Windows XP or SELinux. This work, executed together with Prasad Naldurg, Sriram Rajamani, and John Lambert, was published in [NSRL06] and resulted in the tool NETRA. This contribution does concern authorization but from a quite different perspective than the one discussed in Section 2.3.4, hence its non-inclusion.

- Together with Stefan Haar, Serge Haddad, and Tarek Melliti we considered the problem of *active diagnosis* [HHMS13]. Here, one is given a finite-state system with observable and unobservable actions. Diagnosis consists in detecting, without ambiguity, whether a fault has occurred in a partially observed execution. Active diagnosis aims at controlling the system in order to make it diagnosable. We solve the problem with a game-theoretic approach that improves previous results and obtain several complexity results.

- Some other papers with minor contributions on my part were with Javier Esparza and Pierre Ganty on *Locality-based Abstractions* [EGS05] and with Martin Sachenbacher on *Model-based Test Generation using Quantified CSPs* [SS08].

- A very recent paper with Javier Esparza and Loïg Jezequel [EJS13] concerns the computation of 'summaries' in a concurrent setting, i.e. in a concurrent system, obtain an automaton that describes the behaviour of one particular given component in the presence of the others. While such a summary can be computed using elementary automata theory, the resulting algorithm suffers from the state-explosion problem, and we present a solution based on net unfoldings. While this paper fits thematically into Chapter 3, my contribution to it was minor, so I chose not to include it there.

- On a lighter note, two papers with an educational/recreational purpose, written together with Markus Holzer, analyze the complexity the puzzle games Atomix [HS04a] and Reflexion [HS04b], which is shown to range from NL-complete to PSPACE-complete, depending on which elements are used in the puzzle.

Chapter 2

Pushdown systems

Stacks are among the oldest data structures in computer science. The invention of the concept is variously ascribed to Turing [Tur45], Samelson and Bauer [BS57], or Hamblin [Ham57]. In 1959, Newell et al [NSS59] introduced the notion of a *pushdown automaton* (PDA).

A PDA is a machine that can read its input from left to right, has finite memory (called the *control*) and has an additional storage tape, called the *stack*. The stack has infinite storage capacity, but only the most recently added symbol is visible, or can be removed.

The early interest of computer scientists into stacks can be attributed to a number of factors. The languages recognized by PDA are exactly the context-free languages, which were very useful in specifying and building parsers for programming languages [ALSU86]. The semantics of a programming language with procedures can be explained by a machine with a stack, and indeed CPU architectures habitually feature stack registers and related facilities. Programming languages such as Forth and Postscript are expressly built around the concept of a stack machine, and to some extent also the Java Virtual Machine.

Starting in the 1990s, when model-checking started to become a practical possibility, the pushdown model also became interesting for verification: here, one is not interested in the words accepted or generated by the automaton but in its internal behaviour. In this context, we prefer to speak of a pushdown system (PDS).

This interest stems from the fact mentioned above: PDS can conveniently model programs with procedures that may call one another, passing parameters and having global and local variables. Here, the stack is used

to store information about pending call sites. On the theoretical side, it was quickly established that most interesting verification problems, including model-checking μ-calculus, are decidable for PDS [Wal96], however, in an interesting contrast to finite-state systems, branching-time logics are more expensive to model-check than linear-time logics. E.g., when the specification is fixed, model-checking EF is PSPACE-complete [BEM97] and model-checking CTL is EXPTIME-complete [Wal00], whereas the problem for LTL is polynomial [BEM97, FWW97]. On the practical side, efficient algorithms and tools for pushdown-based verification were developed both in academia and in industry [BR00]. These initial successes have sparked a large body of work during the last decade, some of which will be referenced in the remaining parts of this chapter.

This chapter is organized as follows: Section 2.1 recalls basic definitions and facts about pushdown systems. Section 2.2 highlights the research on weighted PDS, whereas other contributions are described more concisely in Section 2.3.

2.1 Basics about pushdown systems

This part gives basic definitions that will be useful throughout the chapter. It also provides some known results on PDS and some examples.

We write $\mathtt{tt}$ and $\mathtt{ff}$ for the boolean values *true* and *false* and $\mathbb{B} := \{\mathtt{tt}, \mathtt{ff}\}$. We assume the usual notations for formal languages. Let S be a set called *alphabet*. A sequence $w = s_1 \cdots s_n$, where $s_1, \ldots, s_n \in S$, is called a *word* over S. The empty word is denoted by ε, and $|w| = n$ denotes the length of w. The set of all words over S is denoted S^*, with $S^+ := S^* \setminus \{\varepsilon\}$. A *language* over S is a subset of S^*.

A *labelled transition system* (LTS) is a tuple $\mathcal{T} = \langle S, A, \rightarrow \rangle$, where S is a set of *states*, A is a set of *actions*, and $\rightarrow \subseteq S \times A \times S$ is a *transition relation*. If $(s, a, s') \in \rightarrow$ we say that the system can move from state s to s' by performing action a and usually write $s \xrightarrow{a} s'$. If $s \xrightarrow{a} s'$ for some a, then s is an *immediate predecessor* of s', and s' is an *immediate successor* of s.

A *path* of $\mathcal{T}$ is a sequence of states $s_0 \ldots s_n$ (where $n \geq 0$) such that there are transitions $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We say that such a path has length n and is labelled by the word $a_1 \ldots a_n \in A^*$. We write $s \xrightarrow{w}{}^* s'$ if there is a path from s to s' labelled by w.

If $s \xrightarrow{w}{}^* s'$ for some $w \in A^*$, then s' is said to be *reachable from* s. Given a

set $S' \subseteq S$, the set $pre^*_{\mathcal{T}}(S') = \{\, s \mid \exists s' \in S', w \in A^* \colon s \xrightarrow{w}{}^* s' \,\}$ contains the *predecessors* of S', and the set $post^*_{\mathcal{T}}(S') = \{\, s \mid \exists s' \in S', w \in A^* \colon s' \xrightarrow{w}{}^* s \,\}$ contains the *successors* of S'.

When we are not interested in the actions of an LTS, we use the simpler notion of an (unlabelled) transition system $\langle S, \rightarrow \rangle$. All the relevant notions from this subsection are defined analogously for unlabelled transition systems; we drop the labels from the reachability relations and write $s \rightarrow s'$ and $s \rightarrow^* s'$, respectively.

Definition 2.1 *A finite automaton is a quintuple $\mathcal{M} = \langle Q, \Gamma, \rightarrow, Q_0, F \rangle$ such that Q is a finite set of states, Γ is a finite set called the* input alphabet*, $\rightarrow \subseteq (Q \times \Gamma \times Q)$ is a set of* transitions*, $Q_0 \in Q$ is a set of* initial state*, and $F \subseteq Q$ is the set of* final states*. The language $L(\mathcal{M})$ accepted by $\mathcal{M}$ is the set of all finite words w such that $q_0 \xrightarrow{w}{}^* q_f$ holds for a pair $q_0 \in Q_0$ and $q_f \in F$ in the LTS $\langle Q, \Gamma, \rightarrow \rangle$. A language is called* regular *if and only if there is a finite automaton which accepts it.*

2.1.1 Pushdown systems and reachability

A pushdown automaton is, intuitively, a finite automaton with an additional stack.that may contain a word over some finite *stack alphabet*; its length is unbounded. In the context of this document, we are rarely interested in the language generated by a PDA but in its internal behaviour, so we consider a variant without acceptance condition called PDS.

Definition 2.2 *A* labelled pushdown system *(labelled PDS) is a tuple $\mathcal{P} = \langle P, \Gamma, A, \Delta \rangle$, where P is a finite set of* control locations*, Γ is a finite* stack alphabet*, and A a set of actions. A* configuration *of $\mathcal{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted by $Conf(\mathcal{P})$.*

Δ is a finite subset of $P \times \Gamma \times A \times P \times \Gamma^*$ called the rules*; if $\langle p, \gamma, a, p', w \rangle \in \Delta$, we also write $\langle p, \gamma \rangle \xhookrightarrow{a}_{\mathcal{P}} \langle p', w \rangle$. This determines a step relation $\mathcal{T}_{\mathcal{P}}$ as follows:*

$$\text{If } \langle p, \gamma \rangle \xhookrightarrow{a}_{\mathcal{P}} \langle p', w \rangle, \text{ then } \langle p, \gamma w' \rangle \xRightarrow{a}_{\mathcal{P}} \langle p', w w' \rangle \text{ for all } w' \in \Gamma^*.$$

The LTS associated with $\mathcal{P}$ then is $\mathcal{T}_{\mathcal{P}} = \langle Conf(\mathcal{P}), A, \Rightarrow_{\mathcal{P}} \rangle$.

In the following, we write $pre^*_{\mathcal{P}}$ to mean $pre^*_{\mathcal{T}_{\mathcal{P}}}$ (same for $post^*$). If the pushdown system in question is understood, we drop the index $\mathcal{P}$ from the relations $\Rightarrow_{\mathcal{P}}$ and $\hookrightarrow_{\mathcal{P}}$, from $pre^*_{\mathcal{P}}$ and $post^*_{\mathcal{P}}$.

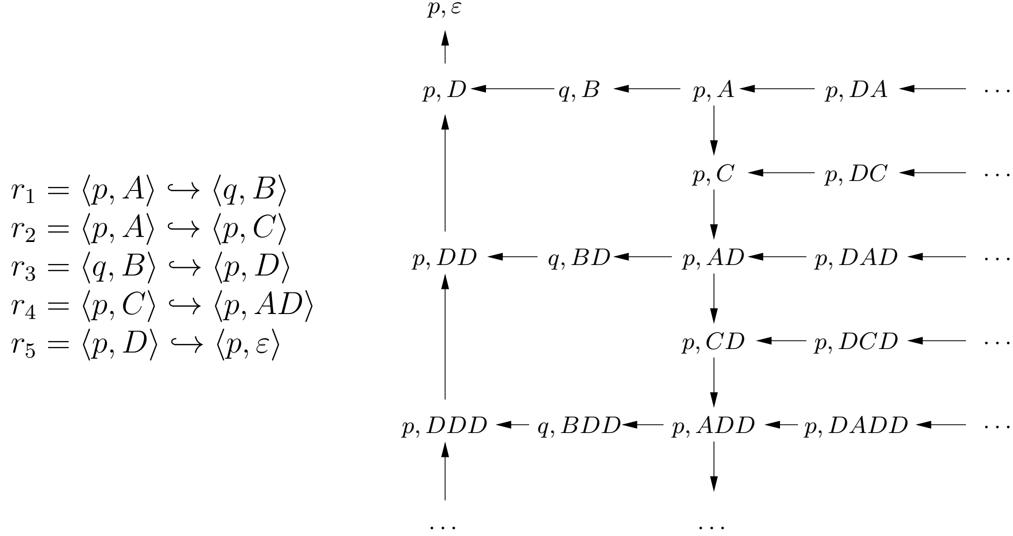

Figure 2.1: A part of the transition system generated by a PDS.

Also, most of the time we consider unlabelled PDS, i.e. without actions; we then write $\mathcal{P} = \langle P, \Gamma, \Delta \rangle$ and omit actions from all arrow relations.

Example 2.1 *Figure 2.1 shows, on the left-hand side, the rules of a PDS $\mathcal{P}$ with two control locations p, q, stack alphabet $\{A, B, C, D, E\}$; on the right-hand side, an excerpt of the (infinite) transition system $\mathcal{T}_\mathcal{P}$ is shown.*

The basic building block for most verification algorithms is reachability analysis. Given a pushdown system $\mathcal{P}$ and a set $\mathcal{C} \subseteq Conf(\mathcal{P})$, a backward (resp. forward) reachability analysis consists of computing the predecessors of elements of $\mathcal{C}$, i.e. the set $pre^*(\mathcal{C})$ (resp. $post^*(\mathcal{C})$).

In general, $\mathcal{C}$ can be an infinite set. But even when $\mathcal{C}$ is finite, the sets $pre^*(\mathcal{C})$ or $post^*(\mathcal{C})$ may be infinite. For instance, in Figure 2.1, all configurations of the form $\langle p, D^n \rangle$, for $n \geq 0$, are predecessors of the single configuration $\langle p, \varepsilon \rangle$. So we need a concise representation of infinite configuration sets, and we use specialized finite automata for this purpose.

Definition 2.3 *Let $\mathcal{P} = \langle P, \Gamma, \Delta \rangle$ be a pushdown system. A $\mathcal{P}$-*automaton *is a finite automaton $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$, whose input alphabet is Γ and whose set of initial states is P. $\mathcal{P}$* accepts *or* recognizes *a configuration $\langle p, w \rangle$ if $p \xrightarrow{w}{}^* q_f$ for some state $q_f \in F$ in the LTS $\langle Q, \Gamma, \rightarrow \rangle$. The set of*

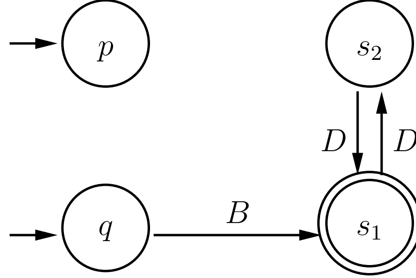

Figure 2.2: A $\mathcal{P}$-automaton.

configurations recognised by a $\mathcal{P}$-automaton $\mathcal{A}$ is denoted by $L(\mathcal{A})$. A set of configurations of $\mathcal{P}$ is regular *if it is recognized by some $\mathcal{P}$-automaton.*

We say that $\mathcal{A}$ is *normalized* if it does not contain any transition whose target is an initial state. Note that any $\mathcal{P}$-automaton can be easily transformed into a normalized automaton accepting the same set of configurations, by adding at most $|P|$ states.

Example 2.2 *Figure 2.2 shows a normalized $\mathcal{P}$-automaton for the PDS $\mathcal{P}$ from Example 2.1. The regular set of configurations accepted by this automaton is $\{ \langle q, BD^{2n} \rangle \mid n \geq 0 \}$.*

A fundamental result discovered by Büchi [Büc64] and Caucal [Cau92] is that regularity is preserved by forward and backward reachability.

Theorem 2.1 *[Büc64, Cau92] Let $\mathcal{P}$ be a PDS and $\mathcal{C}$ a regular subset of $Conf(\mathcal{P})$. Then $pre^*(\mathcal{C})$ and $post^*(\mathcal{C})$ are also regular.*

Moreover, the result is constructive: given a normalized $\mathcal{P}$-automaton $\mathcal{A}$ accepting set $\mathcal{C}$, we can transform $\mathcal{A}$ into an automaton $\mathcal{A}'$ that accepts either $pre^*(\mathcal{C})$ or $post^*(\mathcal{C})$. We give the procedure for pre^*, the one for $post^*$ being similar. Initially, we set $\mathcal{A}' := \mathcal{A}$. Then we add transitions to $\mathcal{A}'$, according to the rule below, until no more transitions can be added. The rule is as follows:

> If $\langle p, \gamma \rangle \hookrightarrow \langle p', w' \rangle$ is a rule of $\mathcal{P}$ and $p' \xrightarrow{w'}{}^* q$ currently holds in $\mathcal{A}'$, then add the transition $p \xrightarrow{\gamma} q$ (where q is any state of $\mathcal{A}'$).

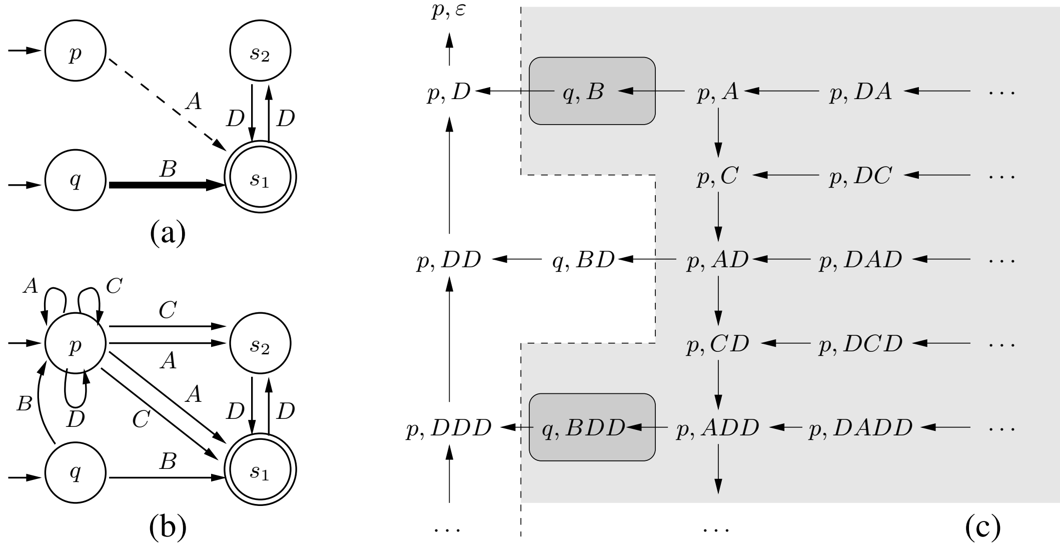

Figure 2.3: (a) Application of *pre** rule to $\langle p, A\rangle \hookrightarrow \langle q, B\rangle$; (b) Final automaton for *pre*$^*(\mathcal{C})$, where $\mathcal{C} = \{\, \langle B, BD^{2n}\rangle \mid n \geq 0\,\}$; (c) *pre*$^*(\mathcal{C})$ in shaded area, elements of $\mathcal{C}$ in darker shade.

Example 2.3 *Consider the PDS $\mathcal{P}$ from Figure 2.1 and the $\mathcal{P}$-automaton $\mathcal{A}$ from Figure 2.2, which accepts the set $\mathcal{C} := \{\, \langle q, BD^{2n}\rangle \mid n \geq 0\,\}$. Figure 2.3 (a) shows an application of the saturation rule to that automaton: due to the rule $\langle p, A\rangle \hookrightarrow \langle q, B\rangle$, the transition $\langle q, B, s_1\rangle$ (shown in bold) induces another transition $\langle p, A, s_1\rangle$ (dotted line), which will be added to the automaton. Note that s_1 could be any state of $\mathcal{A}$, not just a final state. The final result $\mathcal{A}'$, i.e. the fixpoint of this iteration, is shown in Figure 2.3 (b) and accepts the shaded area of the transition graph shown in Figure 2.3 (c). Indeed, for instance $\langle p, DC\rangle$ is accepted in $\mathcal{A}'$ by the path $p \xrightarrow{D} p \xrightarrow{C} s_1$ and is the predecessor of $\langle q, BDD\rangle \in \mathcal{C}$ via rules r_5, r_4, r_2, r_4, r_1, in that order.*

In 1997, both Bouajjani et al [BEM97] and Finkel et al [FWW97] gave cubic algorithms for computing $\mathcal{A}'$. A more precise analysis in [EHRS00] gave $\mathcal{O}(|Q|^2 \cdot |\Delta|)$, where Q are the states of $\mathcal{A}$ and Δ the rule set of $\mathcal{P}$.

The idea for *post** is analogous: for any rule $\langle p, \gamma\rangle \hookrightarrow \langle p', w'\rangle$, one checks whether the left-hand side is 'read' by the current automaton $\mathcal{A}'$, i.e. $p \xrightarrow{\gamma} q$ for some state q, and if so, one adds a path $p' \xrightarrow{w'}{}^* q$ to $\mathcal{A}'$. The catch is that w can be of variable length, so one has to insert either an ε-labelled transition, one single transition, or multiple transitions, which requires to introduce

additional states. It is shown in [EHRS00] that the number of required additional states is bounded by $|\Delta|$. So although the *post** algorithm follows the same idea as *pre**, it has some additional complications that render it more cumbersome.[1]

Definition 2.4 *Let $\mathcal{P} = \langle P, \Gamma, \Delta\rangle$ be a PDS. We call $\mathcal{P}$* normalized *if $|w| \leq 2$ for all rules $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$. A tuple $\langle p, \gamma, q\rangle \in P \times \Gamma \times P$ is called* pop triple *if $\langle p, \gamma\rangle \Rightarrow^* \langle q, \varepsilon\rangle$. A set of configurations $\mathcal{C}$ is called* basic *if $\mathcal{C} = F \times \{\varepsilon\}$, for some set $F \subseteq P$. A (normalized) $\mathcal{P}$-automaton $\mathcal{A}$ is called basic if it is of the form $\langle P, \Gamma, \emptyset, P, F\rangle$, i.e. it accepts $\mathcal{C}$, has no transitions, and no states other than $\mathcal{P}$.*

To simplify the presentation, we argue that any reachability problem for PDS can be reduced to determining the pop triples of another PDS, using backwards reachability. For this, it suffices to realize that (1) any PDS can be normalized; (2) forward reachability can be reduced to backward reachability; (3) it suffices to consider basic automata. All transformations are folklore, we just give some pointers. For the following, fix a PDS $\mathcal{P} = \langle P, \Gamma, \Delta\rangle$ and a regular set $\mathcal{C} \subseteq \mathit{Conf}(\mathcal{P})$.

1. The details of this transformation are given in [Sch02b], Theorem 3.1. It consists in breaking up rules with long right-hand sides into multiple rules via 'intermediate' configurations. We will from now on assume that all PDS are normalized.

2. For this reduction, one can construct a PDS $\mathcal{P}^{-1}$ that executes the rules of $\mathcal{P}$ 'in reverse'. Such a construction is given, e.g., in [KSSK09b], Appendix C. Notice that this reduction also covers the case of *weighted* PDS that will be discussed in Section 2.2.

3. The reduction extends $\mathcal{P}$ with additional 'pop' rules for all transitions of a normalized $\mathcal{P}$-automaton $\mathcal{A}$ accepting $\mathcal{C}$. An execution of this extended PDS first behaves like $\mathcal{P}$, then simulates $\mathcal{A}$ and arrives at a

[1] The complication of having to introduce additional states for *post** arises because the formulation we chose for PDS has asymmetric rules – the length of the left-hand side is fixed, whereas the length of the right-hand side is variable. In the formulation of PDS used in [FWW97], pushdown rules are of the form $q \xrightarrow{a+} q'$ (push) or $q \xrightarrow{a-} q'$ (pop). These rules are symmetric, and the problem does not arise. However, we stick to our formulation because it is more suitable for modelling procedural programs, see Section 2.1.3.

final state of $\mathcal{A}$ iff the simulation began at a configuration from $\mathcal{C}$. The details are given in [RSJM05], Section 3.1.1.

Note that these reductions are not necessarily desirable from an efficiency point of view - in particular they increase the number of control locations, which, as stated before, contribute quadratically to the *pre** running time. Therefore, most papers upon which this document are based, present direct constructions, including for *post**, rather than the reductions given here. The limitation to backwards reachability, chosen in this document, serves to simplify the presentation and showcase more clearly the underlying concepts.

So suppose that we are given $\mathcal{P}, \mathcal{A}, \mathcal{C}$, and let $\mathcal{A}'$ be the automaton obtained through the *pre** computation. From [BEM97] we know the following fact: if p, q are control locations, then $\langle p, \gamma, q\rangle$ is a pop triple iff $p \xrightarrow{\gamma} q$ is a transition of $\mathcal{A}'$. If $\mathcal{A}$ is basic, then all states are control locations, therefore *every* transition signifies a pop triple. This motivates the following remark:

Remark 2.1 *Let $c = \langle q_0, \gamma_1, \ldots, y_n\rangle$ be a configuration. If $\mathcal{A}, \mathcal{C}$ are basic, with $\mathcal{A} = \langle P, \Gamma, \emptyset, P, F\rangle$, then $c \in pre^*(\mathcal{C})$ iff there exist $q_1, \ldots, q_{n-1} \in P$ and $q_n \in F$ such that $\langle q_{i-1}, \gamma\rangle \Rightarrow^* \langle q_i, \varepsilon\rangle$ holds for $i = 1, \ldots, n$. In other words, any path from c to $\mathcal{C}$ can be decomposed into n sub-paths, each of which corresponds to a pop triple and hence a transition $q_{i-1} \xrightarrow{\gamma_i} q_i$, for $i = 1, \ldots, n$, of $\mathcal{A}'$.*

Remark 2.1 gives rise to a different characterization of the *pre** algorithm in terms of an equation system. Let $X := \{\, [\![p, \gamma, q]\!] \mid p, q \in P,\ \gamma \in \Gamma \,\}$ be a set of boolean-valued variables, where $[\![p, \gamma, q]\!]$ has the meaning "$\langle p, \gamma, q\rangle$ is a pop triple". We associate an equation with each variable, where $[\![p, \gamma, q]\!]^?$ stands for a symbolic constant evaluating to true iff $\langle p, \gamma\rangle \hookrightarrow \langle q, \varepsilon\rangle \in \Delta$:

$$[\![p, \gamma, q]\!] = [\![p, \gamma, q]\!]^? \vee \bigvee_{\langle p,\gamma\rangle \hookrightarrow \langle r,\gamma'\rangle} [\![r, \gamma', q]\!] \vee \bigvee_{\substack{\langle p,\gamma\rangle \hookrightarrow \langle r,\gamma'\gamma'' p\rangle \\ r' \in Q}} [\![r, \gamma, r']\!] \wedge [\![r', \gamma'', q]\!] \quad (2.1)$$

Intuitively, (2.1) simply lists the all possible cases how $\langle p, \gamma\rangle$ could be transformed into $\langle q, \varepsilon\rangle$. Thus, $\langle p, \gamma, q\rangle$ is a pop triple iff $[\![p, \gamma, q]\!]$ is true in the greatest (in the sense of $\texttt{ff} > \texttt{tt}$) solution of the equation system. The *pre** algorithms of [BEM97, FWW97, EHRS00] can then be seen as an efficient way to solve the equation system (2.1). We shall come back to this analogy to explain the contributions in Section 2.2.2.

2.1.2 Related constructions

We mention some similar constructions on related models. To begin with, the *pre** computation can be generalized to **context-free grammars**. Here, the predecessor relation is defined over sentence forms (containing both variables and terminals) via productions, i.e., $\alpha A \gamma$ is a predecessor of $\alpha\beta\gamma$ if there exists a production $A \to \beta$. Book and Otto showed that backwards reachability preserves regularity in this case [BO93], and an analogous automata-theoretic construction takes $\mathcal{O}(n \cdot k^3)$ time, where n is the size of the productions and k the number of automaton states [ERS00]. Of course, forward reachability does not preserve regularity in this case.

Bouajjani et al [BEM97] presented an extension to **alternating PDS**. The basic idea is that pushdown rules can have a collection of right-hand sides, e.g. $\langle p, \gamma \rangle \hookrightarrow \{\langle p_1, \gamma_1 \rangle, \langle p_2, \gamma_2 \rangle\}$, and an execution of the rule creates one branch for each element of the right-hand side. Then, a configuration c is considered to be a predecessor of a set $\mathcal{C}$ if all branches of an execution starting at c end up in an element of $\mathcal{C}$. The resulting reachability problem can be solved by extending the construction of Section 2.1.1 to alternating automata.

Löding [Löd06] extended the principle to **ground tree rewrite systems** (GTRS). Here, the equivalent of a configuration is a labelled tree, and transition rules describe how one subtree can be replaced by another. Regular sets of trees can be described by bottom-up tree automata (TA), and a reachability algorithm transforms on TA into a TA recognizing the set of predecessor trees. The diploma thesis of Andreas Gaiser [Gai08], supervised by Javier Esparza and myself, describes an implementation of this technique with an application to the analysis of functional programs.

Ong and Hague [HO07] have generalized the reachability analysis for **higher-order PDS**, in which configurations can include stacks, stacks of stacks, etc. More recently, this was extended to **collapsible** PDS [BCHS12].

Finally, let us remark that the reachability relation between PDS configurations can be captured by a **transducer**. In a nutshell, this is because, for any pair of configurations such that $\langle p, w \rangle \Rightarrow^* \langle p', w' \rangle$, the path leading from one to the other can be decomposed into a 'pop phase' and a 'push phase'. I.e., there exists a decomposition $w = w_1 \gamma w_2$, $w' = w_3 w_2$ and a location q such that $\langle p, w_1 \rangle \Rightarrow^* \langle q, \varepsilon \rangle$ and $\langle q, \gamma \rangle \Rightarrow^* \langle p', w_3 \rangle$. Thus, it suffices to determine the pop triples of $\mathcal{P}$ and $\mathcal{P}^{-1}$. Such a construction was first done by Caucal [Cau92].

2.1.3 Modelling programs as PDS

We conclude the introduction by recalling how PDS can be used to model sequential programs with procedures, which constitutes the principal motivation for studying this model. A more extensive treatment is found, e.g., in [Sch02b], Section 2.3. The transformation of procedural programs into PDS exploited by the model checker Moped, which implements a BDD-based variant of pushdown reachability.

Programs written in languages such as C or Java are typically composed of a number of procedures. Procedures may call each other, possibly recursively, passing argument values to the callee or returning values to the caller.

Let us first discuss how to model the control flow of such a program, which in this case can be seen as a set of flow graphs, where edges are annotated with the actions of that program, eventually including calls and returns. In this case, we set the stack alphabet to $\Gamma := N$, the set of nodes in all flowgraphs, and P is a singleton with a dummy element $(\cdot)$. Calls/returns then naturally correspond to push/pop rules, e.g., a rule

$$\langle \cdot, n \rangle \hookrightarrow \langle \cdot, n'n'' \rangle$$

corresponds to an edge from n to n'' annotated with a call to some procedure that starts at node n'. Notice that the restriction to right-hand sides of length 2, introduced in Section 2.1.1, is natural here.

Now suppose that we wish to add data to the model. Programming languages typically offer global variables, accessible to all procedures, and local variables, newly instantiated whenever a procedure is invoked and accessible only to that invocation. Both types of variables can be taken into account faithfully by encoding them into the control locations and the stack, respectively. Let G be the valuations of the global variables and L those of the locals (w.l.o.g., we assume a uniform set of variables over all procedures, for simplicity). Then we set $P := G$ and $\Gamma := N \times L$. A call statement now has the form

$$\langle g, \langle n, \ell \rangle \rangle \hookrightarrow \langle g', \langle n', \ell' \rangle \langle n'', \ell'' \rangle \rangle,$$

Note how a call statement generates a new set of locals and a return statement destroys them, while global variables are preserved. The chosen modelling implies a limitation to finite data types, since P and Γ must be finite.

```
bool g=true;
void main()
begin
  n0: level1();
  n1: level1();
  n2: assume(g);
  n3: stop;
end
```

```
void leveli()
begin
  n4: leveli+1();
  n5: leveli+1();
  n6: return;
end
```

```
void leveln()
begin
  n7: g := not g;
  n8: return;
end
```

Figure 2.4: Simple program with one boolean variable, where $i = 1, \ldots, n-1$.

Example 2.4 *Consider the program in Figure 2.4, which is a variation of a an example from [BR00]. The program consists of a main function and functions* $\texttt{level}_i$*, for* $i = 1, \ldots, n$*. The program has one global boolean variable* g*, so we set* $P = G = \mathbb{B}$*. There are no local variables, so we would just set* $\Gamma := \{n_0, \ldots, n_8\}$*. For instance, the statement at line* n_0 *would translate to two rules of the form* $\langle b, n_0 \rangle \hookrightarrow \langle b, n_4 n_1 \rangle$*, for* $b \in \mathbb{B}$*.*

The program in Figure 2.4 does not have any recursive procedure calls. So it could in principle be handled by a finite-state model checker that 'inlines' procedure calls. However, the resulting program would be of size $\mathcal{O}(2^n)$. On the other hand, pushdown reachability can check whether it is possible for the assumption in *main* to fail; this amounts to testing whether $\langle \texttt{tt}, n_0 \rangle \in pre^*(\{\langle \texttt{ff}, n_2 \rangle\})$, which takes $\mathcal{O}(n)$ time.

As this example shows, the interest of PDS in verification is twofold: first, PDS can seamlessly handle recursive procedure calls, which finite-state model checkers cannot handle faithfully; secondly, PDS model checking can be more efficient even in the absence of recursion.

Finally, we remark that PDS are expressively equivalent to Boolean Programs, which were introduced as an abstract domain for model-checking device drivers inside the SLAM toolkit [BR00], and Recursive State Machines [AEY01, BGR01]. The latter allow to model programs as a collection of finite-state machines that may invoke one another; they thus have an explicit notion of procedure but no explicit separation of control location and stack.

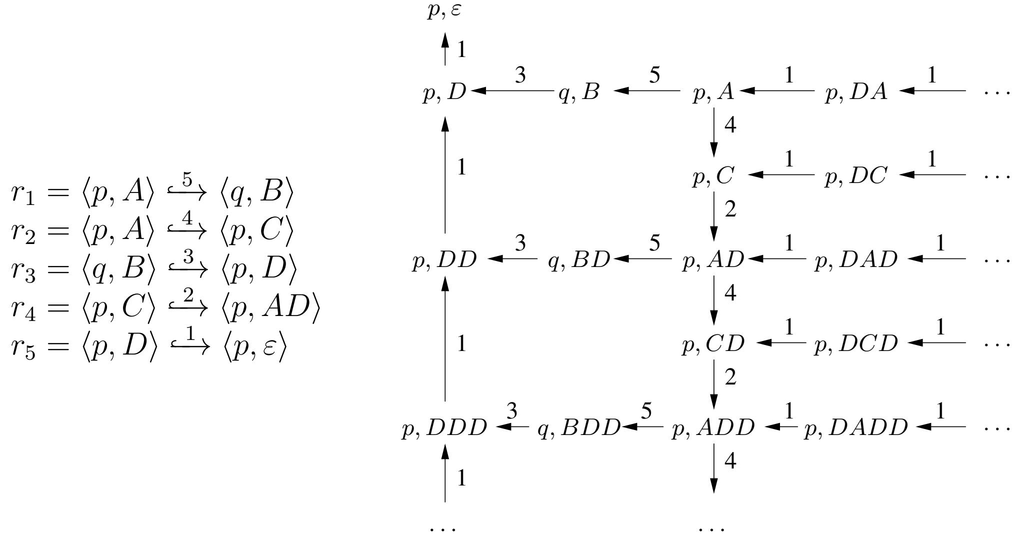

Figure 2.5: Transition graph with integer weights.

2.2 Weighted PDS

This section reports on the first group of contributions to the verification of pushdown systems; it is based on results published in [RSJ03, RSJM05, KSSK09a], presented in two parts.

Section 2.1 summarized reachability analysis on unweighted PDS. The answer given by, e.g., a *pre** query, is *qualitative*: it allows to answer whether a configuration can reach some target or not. An interesting extension is to ask *quantitative* questions such as "What is the minimum number of steps to reach a given configuration?" or "What is the least costly/most secure etc execution satisfying a certain property?" We first develop some definitions, then present results for the case of *bounded* (Section 2.2.1) and *unbounded* semirings (Section 2.2.2). That framework allows to answer the aforementioned questions and has wide-ranging applications for data-flow analysis and PDS-based verification, discussed in Section 2.2.3.

Example 2.5 *Let us reconsider the PDS from Figure 2.1 and suppose that each execution of a rule incurs a certain cost, e.g. a non-negative integer number. The costs would naturally be annotated on the rules and give rise to a* weighted *transition system. Figure 2.5 shows an example, where the cost of going from $\langle p, DA \rangle$ to $\langle p, D \rangle$ is either* $1 + 5 + 3 = 9$ *(on the direct path),*

or generally $9 + 7i$ when going through $\langle q, BD^i \rangle$. In this case, it is natural to ask for the minimal cost of going from one given configuration to another, as well as for a path realizing that cost.

Example 2.6 *Consider the same PDS, but suppose that we want to know whether a 'bad' action (say, r_4) occurs on some, all, or none of the paths connecting a pair of configurations c, c'. Also, we would be interested in examples for both cases, where applicable. E.g., for $c = \langle p, DA \rangle$ and $c' = \langle p, D \rangle$, the paths via $\langle q, B \rangle$ and $\langle q, BD \rangle$, respectively, would provide the desired answer.*

In both cases, we need an operator to *aggregate values along a path* (addition in Example 2.5, logical or in Example 2.6) and another to produce *summaries of multiple paths* (e.g., min in Example 2.5). The domain is totally ordered in the first case (so only one witness path is needed), or partially ordered in the second case (requiring multiple witnesses). This motivates the following definition:

Definition 2.5 *An* idempotent semiring *is a tuple $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$, where D is a set of* weights*, called the* domain *of $\mathcal{S}$; $\bar{0}$ and $\bar{1}$ are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with $\bar{0}$ as its neutral element, and where $\oplus$ is idempotent (i.e., for all $a \in D$, $a \oplus a = a$).*

2. *$(D, \otimes)$ is a monoid with the neutral element $\bar{1}$.*

3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \quad \text{and} \quad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$

4. *$\bar{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $\bar{0} \otimes a = \bar{0} = a \otimes \bar{0}$.*

For all $a, b \in D$, we write $a \sqsubseteq_{\mathcal{S}} b$ iff $a \oplus b = a$ and $a \sqsubset_{\mathcal{S}} b$ if $a \sqsubseteq b$ and $a \neq b$, omitting the index $\mathcal{S}$ when it is understood. We say that the monoid $\langle D, \oplus \rangle$ (and by extension $\mathcal{S}$) is bounded *if the partial order $\sqsubseteq$ does not contain an infinite descending chain, i.e. no infinite sequence $a_1 \sqsupset a_2 \sqsupset a_3 \sqsupset \cdots$.*

Intuitively, the operator $\otimes$ is used to connect values along a path, whereas $\oplus$ computes a summary of multiple values. The annihilator $\bar{0}$ has the meaning of 'no path exists'. Assuming its existence is no real restriction: a semiring satisfying (1)–(3), where $\bar{0}$ is not an annihilator, can be extended with an artificial annihilator $\bar{0}'$, for which we define $\bar{0}' \otimes a = \bar{0}' = a \otimes \bar{0}'$ and $\bar{0}' \oplus a = a$ for all $a \in D \cup \{\bar{0}'\}$; in this case, $\bar{0}'$ is the new neutral element of $\oplus$, whereas $\bar{0}$ is "almost neutral", except for $\bar{0} \oplus \bar{0}' = \bar{0}$. The issue of distributivity in (3) will be discussed in Section 2.2.1.

Example 2.7 *In Example 2.5, the weights attributed to the rules form part of a bounded idempotent semiring $\mathcal{S}_1 = \langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0\rangle$. In Example 2.6, the weights can be expressed by $\mathcal{S}_2 = \langle 2^{\mathbb{B}}, \cup, \otimes, \emptyset, \{\texttt{ff}\}\rangle$, where $B_1 \otimes B_2 := \{\, b_1 \vee b_2 \mid b_1 \in B_1,\ b_2 \in B_2 \,\}$. Here, a weight $B \subseteq \mathbb{B}$ is interpreted as carrying information about a set of paths. B contains* tt *if at least one of those paths contains the 'bad' action, and* ff *if at least one of them does not. Notice that $\sqsubseteq_{\mathcal{S}_1}$ is a total order, while $\sqsubseteq_{\mathcal{S}_2}$ contains an unordered pair, i.e. $\{\texttt{tt}\}$ and $\{\texttt{ff}\}$. If r_4 is the 'bad' rule, we would equip it with the weight $\{\texttt{tt}\}$ and all others with $\{\texttt{ff}\}$.*

A weighted pushdown system is now simply defined as a PDS equipped with semiring weights.

Definition 2.6 *A* weighted pushdown system *(WPDS) is a tuple $\mathcal{W} = \langle \mathcal{P}, \mathcal{S}\rangle$, where $\mathcal{P} = \langle P, \Gamma, D, \Delta\rangle$ is a labelled PDS, whose actions are the weights of idempotent semiring $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1}\rangle$. For a word $\sigma = d_1 \cdots d_n \in D^*$, let $val_{\mathcal{S}}(\sigma) = d_1 \otimes \cdots \otimes d_n$ its associated semiring value; by definition $val_{\mathcal{S}}(\varepsilon) := \bar{1}$.*

Remark 2.2 *In the complexity analyses to follow, we will assume that operations $\oplus$ and $\otimes$ can be carried out in $\mathcal{O}(1)$ time, or more precisely in a time that depends only on the semiring itself but not on the WPDS using it.*

We are now in a position to formalize the problems mentioned earlier. As explained in Section 2.1, this presentation focuses on backwards reachability, while [RSJM05, KSSK09b] give specialized treatments for both forward and backward analysis.

Definition 2.7 *Let $\mathcal{W} = \langle \mathcal{P}, \mathcal{S}\rangle$ be a WPDS with $\mathcal{P} = \langle P, \Gamma, D, \Delta\rangle$ and $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1}\rangle$, and let $\mathcal{C} \subseteq Conf(\mathcal{P})$ be a regular set. The* generalized

predecessor problem *(GPP) for $\mathcal{W}, \mathcal{C}$ is to compute, for each configuration $c \in \mathit{Conf}(\mathcal{P})$, the value*

$$\delta(c) := \bigoplus \{\, val_{\mathcal{S}}(\sigma) \mid c \overset{\sigma}{\Longrightarrow}{}^{*} c',\ c' \in \mathcal{C} \,\}.$$

The GPP problem can be interpreted as a generalized shortest-path problem on an infinite graph with multiple targets ($\mathcal{C}$) and all configurations as sources, where $\delta(c)$ gives the 'distance' from c to $\mathcal{C}$.

Again, we need automata to deal with infinite configuration sets:

Definition 2.8 *Let $\mathcal{W} = \langle \mathcal{P}, \mathcal{S} \rangle$ be a WPDS with $\mathcal{P} = \langle P, \Gamma, D, \Delta \rangle$ and $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1} \rangle$. A finite automaton $\mathcal{A} = \langle Q, \Gamma \times D, \rightarrow, P, F \rangle$ is called $\mathcal{W}$-automaton; for convenience, we will denote a transition $\langle s, \langle \gamma, d \rangle, s' \rangle$ as $s \xrightarrow[d]{\gamma} s'$ or $s \xrightarrow{\gamma(d)} s'$. We say that $\mathcal{A}$ accepts configuration $c = \langle p, w \rangle \in \mathit{Conf}(\mathcal{P})$ with weight*

$$val_{\mathcal{A}}(c) := \bigoplus \{\, val_{\mathcal{S}}(\sigma) \mid p \xrightarrow[\sigma]{w}{}^{*} q,\ q \in F \,\}.$$

Other notions of $\mathcal{P}$-automata will be silently used for $\mathcal{W}$-automata when their meaning is clear. To obtain a basic $\mathcal{W}$-automaton from an arbitrary $\mathcal{W}$-automaton, one uses the same procedure as on page 21, except that the new rules have weight $\bar{1}$. In the computation of $val_{\mathcal{A}}(c)$, we use $\bigoplus \emptyset = \bar{0}$.

Example 2.8 *Consider the automaton shown in Figure 2.6 (a), which is a $\mathcal{W}$-automaton for the WPDS shown in Figure 2.5 with semiring $\mathcal{S}_1$. The automaton accepts the configurations from set $\{\, \langle q, BD^{2n} \rangle \mid n \geq 0 \,\}$ with weight 0 and all others with weight ∞.*

2.2.1 Bounded semirings

In this section we solve the GPP for the case of idempotent semirings that are *bounded*. We first discuss the computation of $\delta(c)$, then the generation of *witnesses* for those values. Afterwards, we discuss some related issues such as the distributivity condition, how to use the framework in a symbolic model checker, and a differential variant. A more extensive discussion of applications and related work is contained in Section 2.2.3. Unless otherwise stated, the results in this paper stem from work done with Thomas Reps, Somesh Jha, and David Melski [RSJ03, RSJM05].

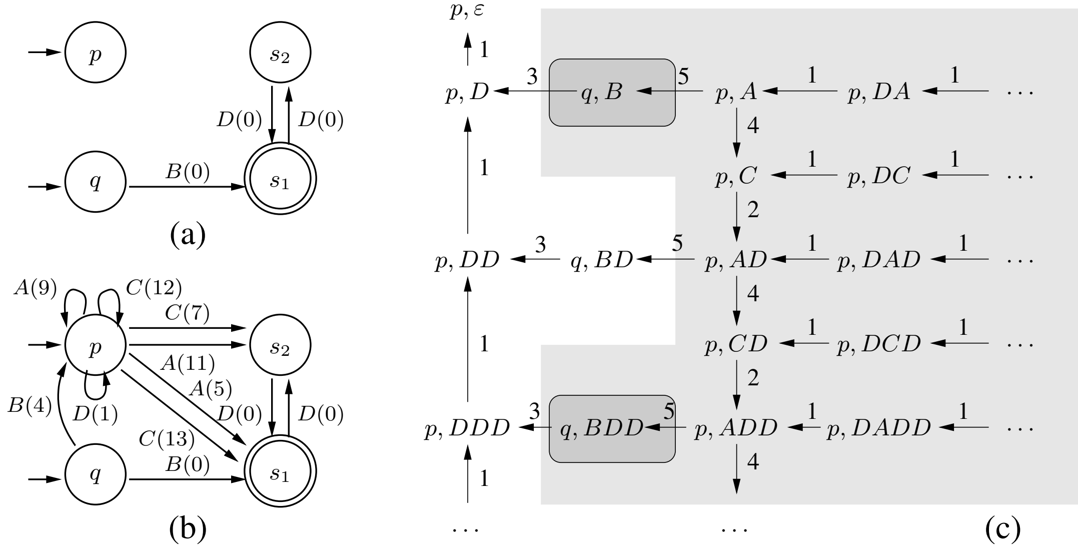

Figure 2.6: (a) $\mathcal{W}$-automaton accepting $\mathcal{C} := \{ \langle q, BD^{2n} \rangle \mid n \geq 0 \}$ with weight 0; (b) automaton accepting each configuration with $\delta(c)$; (c) PDS transition graph with weights, $\mathcal{C}$ and $pre^*(\mathcal{C})$ indicated by shading.

In this section, fix a WPDS $\mathcal{W} = \langle \mathcal{P}, \mathcal{S} \rangle$, where $\mathcal{P} = \langle P, \Gamma, D, \Delta \rangle$ and $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1} \rangle$ is bounded. Let $\mathcal{C}$ be a basic set of configurations. Then the GPP problem for $\mathcal{W}, \mathcal{C}$ can be solved by extending the saturation method for PDS. Again, we first construct an automaton $\mathcal{A}$ for $\mathcal{C}$ and then transform it into another automaton $\mathcal{A}'$ by adding transitions and modifying their weights, until we reach a fixpoint. The saturation rule is given in Figure 2.7.

The following proposition summarizes results from [RSJM05], Section 3.1:

Proposition 2.1 *Algorithm 1 terminates, and the resulting automaton $\mathcal{A}'$ accepts each configuration $c \in Conf(\mathcal{P})$ with weight $\delta(c)$. Moreover, the algorithm can be implemented in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot \ell)$ time (cf. Remark 2.2), where ℓ is the length of the longest descending chain starting at a value that appears in $\mathcal{A}'$ during the computation.*

Notes on the proof In [RSJM05], the proof is through a reduction to abstract grammar problems [Ram96]. The algorithm is implemented through a modification of the pre^* algorithm for PDS. Termination is guaranteed

Algorithm 1

Input: WPDS $\mathcal{W} = \langle \mathcal{P}, \mathcal{S} \rangle$, basic $\mathcal{W}$-automaton $\mathcal{A}$ accepting $\mathcal{C}$
Output: $\mathcal{W}$-Automaton $\mathcal{A}'$ representing $\delta(c)$ for all $c \in \mathit{Conf}(\mathcal{P})$.

Set $\mathcal{A}' := \mathcal{A}$; then apply the following rule until fixpoint is reached:

If $\langle p, \gamma \rangle \overset{d}{\hookrightarrow} \langle p', w' \rangle$ is a rule and $p' \xrightarrow[\sigma]{w'}{}^* q$ holds in $\mathcal{A}'$, then:

(i) let $d' := \mathit{val}_{\mathcal{S}}(d\sigma)$;

(ii) if $\mathcal{A}'$ contains a transition $p \xrightarrow[d'']{\gamma} q$, replace it by $p \xrightarrow[d' \oplus d'']{\gamma} q$;

(iii) if no such transition exists, add $p \xrightarrow[d']{\gamma} q$.

Figure 2.7: Saturation rule for solving GPP for bounded semirings.

because every transition can change its value at most a finite number of times in a bounded semiring, hence the complexity increase (w.r.t. PDS) by a factor of ℓ. Note that ℓ is well-defined because (i) there are at most $k := |P| \cdot |\Delta|$ transitions in $\mathcal{A}'$; (ii) the weight of a transition can only decrease from its initial weight; (iii) the initial weight can be a $\otimes$-product of at most k rule labels.

Example 2.9 *We continue from Example 2.8 with the automaton $\mathcal{A}$ in Figure 2.6 (a). To obtain a basic automaton, we first replace its transitions by these 'artificial' WPDS rules:*

$$r'_1 := \langle q, B \rangle \overset{0}{\hookrightarrow} \langle s_1, \varepsilon \rangle;$$
$$r'_2 := \langle s_1, D \rangle \overset{0}{\hookrightarrow} \langle s_2, \varepsilon \rangle;$$
$$r'_3 := \langle s_2, D \rangle \overset{0}{\hookrightarrow} \langle s_1, \varepsilon \rangle.$$

The result of applying Algorithm 1 is shown in Figure 2.6 (b). In this automaton, every configuration happens to be accepted by at most one path, so we can easily read off the results. For instance $\langle p, DC \rangle$ is accepted in $\mathcal{A}'$ by the path $p \xrightarrow[1]{D} p \xrightarrow[13]{C} s_1$, and indeed, the shortest path to an element of $\mathcal{C}$ is to $\langle q, BDD \rangle \in \mathcal{C}$ via rules r_5, r_4, r_2, r_4, r_1 (and from there to $\langle s_1, \varepsilon \rangle$ via r'_1, r'_2, r'_3), whose total weight is 14.

The procedure can be extended to recover *witnesses* at the same time, which explain the value $\delta(c)$, i.e. for each configuration c a set $\omega(c) \subseteq D^*$

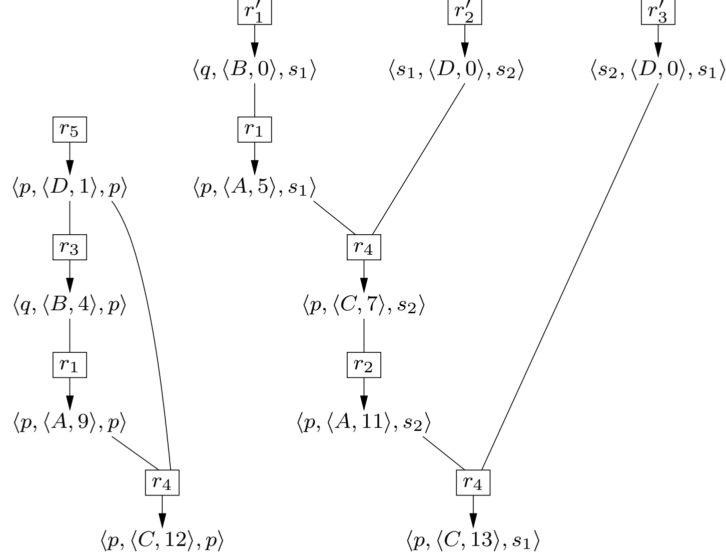

Figure 2.8: Witness graph $\mathcal{G}$ for recovering GPP witnesses in Figure 2.6.

such that $\bigoplus\{ val_{\mathcal{S}}(\sigma) \mid \sigma \in \omega(c) \} = \delta(c)$. Notice that $|\omega(c)|$ may be larger than one (Example 2.6), but is guaranteed to be finite because $\mathcal{S}$ is bounded.

Due to Remark 2.1, it suffices to identify witnesses for every pop triple and concatenate them. Roughly speaking, this is achieved by generating a directed acyclic hypergraph $\mathcal{G}$, called the *witness graph*, that keeps track of the rules and automata transitions involved in every iteration of Algorithm 1. The nodes of $\mathcal{G}$ are all weighted transitions generated during the algorithm (including those that are replaced under case (ii) of the saturation rule). A hyperedge with a set of zero or more sources T, label $r \in \Delta$, and target t records that rule r and the elements of T were used to generate t (see [RSJM05], Section 3.1.4 for details).

Example 2.10 *Figure 2.8 shows the witness graph $\mathcal{G}$ for the previous example. For instance, $p \xrightarrow[7]{C} s_2$ is generated by using rule $r_4 = \langle p, C\rangle \xhookrightarrow{2} \langle p, AD\rangle$ with $p \xrightarrow[5]{A} s_1 \xrightarrow[0]{D} s_2$. Thus a witness for, e.g., $\langle p, DC\rangle$ can be obtained by following the hyperedges backwards and from left to right. For $p \xrightarrow[1]{D} p$ we obtain just r_5, for $p \xrightarrow[13]{C} s_1$ we obtain (from bottom to top) $r_4 r_2 r_4 r_1 r'_1 r'_2 r'_3$, where the latter three rules are artificial and can be ignored.*

When the semiring is not totally ordered, the weight of a configuration c may not be justifiable by a single path, i.e. $|\omega(c)| \geq 2$. In this case, a node

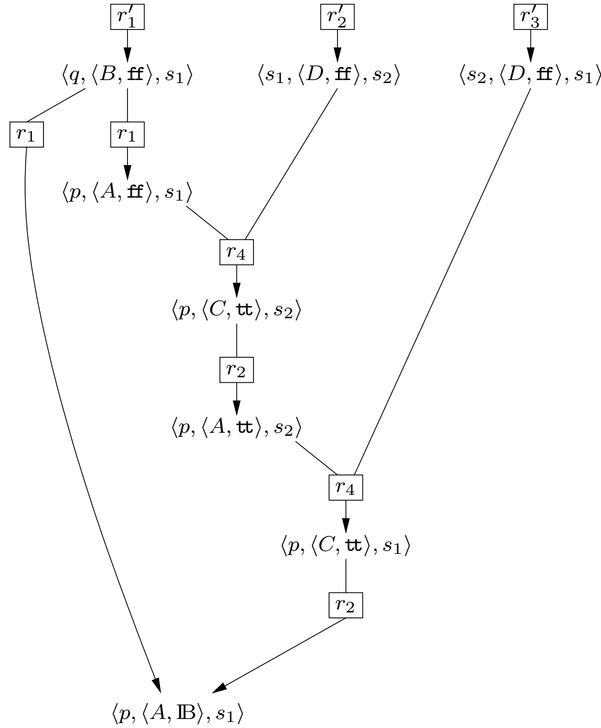

Figure 2.9: Part of the witness graph for $\mathcal{S}_2$; braces omitted around $\texttt{tt}, \texttt{ff}$.

of Γ may have multiple incoming arcs.

Example 2.11 *Figure 2.9 shows a part of the witness graph when $\mathcal{S}_2$ (see Example 2.7) is used instead of $\mathcal{S}_1$. Note that the transition at the bottom, $p \xrightarrow[\mathbb{B}]{A} s_1$, has two incoming hyperedges. These allow to recover a path with value $\{\texttt{ff}\}$ (i.e. without executing the 'bad' rule r_4) to $\langle q, B\rangle$ as well as a path with value $\{\texttt{tt}\}$ that does execute r_4 (even twice) and ends at $\langle q, BDD\rangle$.*

In general, a witness graph of size n can encode WPDS paths of length exponential in n. This happens, for instance, when the procedure is applied to the program in Example 2.4, where for a given value of n, PDS reachability terminates in $\mathcal{O}(n)$ time, but a witness path from the start of `main` to its end has length in $\mathcal{O}(2^n)$.

We shall round off this part by discussing a couple of important variations and extensions of the framework. Further applications of the framework are discussed in Section 2.2.3.

Distributivity

We give an intuition why distributivity is needed in Definition 2.5. As an example, assume a WPDS with tree rules:

$$\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p, \varepsilon\rangle \qquad \langle p, \gamma'\rangle \overset{b}{\hookrightarrow} \langle p, \varepsilon\rangle \qquad \langle p, \gamma'\rangle \overset{c}{\hookrightarrow} \langle p, \varepsilon\rangle$$

We set $c_1 := \langle p, \gamma\gamma'\rangle$, $c_2 := \langle p, \gamma'\rangle$, and $c_3 := \langle p, \varepsilon\rangle\}$. Then we have two paths from c_1 to c_3, i.e. $c_1 \overset{a}{\Longrightarrow} c_2 \overset{b}{\Longrightarrow} c_3$ and $c_1 \overset{a}{\Longrightarrow} c_2 \overset{c}{\Longrightarrow} c_3$. Accordingly, for the GPP with $\mathcal{C} := \{c_3\}$, the value of $\delta(c_1)$ is $d_1 := (a \otimes b) \oplus (a \otimes c)$ according to Definition 2.7.

Now, Proposition 2.1 claims that we can obtain this value from an automaton $\mathcal{A}'$, which contains one transition per pop triple, i.e. $p \xrightarrow[a]{\gamma} p$ and $p \xrightarrow[b \oplus c]{\gamma'} p$. By Definition 2.8, the weight of c_1 in $\mathcal{A}'$ is $d_2 := a \otimes (b \oplus c)$.

Hence, we need the distributivity laws to ensure that $d_1 = d_2$. (The case for the second distributivity law is made by exchanging γ and γ' in the rules.) In [RSJM05], Section 4.4, we explain that the distributivity constraint can be relaxed to a weaker monotonicity condition for all semiring values a, b, c:

$$a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c) \quad \text{and} \quad (a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$$

In this case, Algorithm 1 will provide 'safe' underapproximations, i.e. provide for each configuration c' a value $d \sqsubseteq \delta(c')$.

Encoding relations

Bounded idempotent semirings (Definition 2.5) are a common framework in dataflow analysis and fixpoint equations. However, Algorithm 1 may function correctly in more general cases. We discuss one such case that was not presented in [RSJM05] but has implicitly been used in Moped all the time.[2]

Suppose that $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1}\rangle$ is an idempotent semiring (not necessarily bounded), where the domain D of our semiring contains three subsets D_0, D_1, D_2 such that

1. $\langle D_0, \oplus\rangle$ is a bounded idempotent commutative sub-monoid with neutral element $\bar{0}$;

2. for any $d_1 \in D_1$, $d_0 \in D_0$ we have $d_1 \otimes d_0 \in D_0$;

[2] A presentation similar to this one was included in [Suw09], Section 4.3.1, under the heading "Semiring with locals".

3. for any $d_2 \in D_2$, $d_0, d_0' \in D_0$ we have $d_2 \otimes d_0 \otimes d_0' \in D_0$.

Moreover, let $\mathcal{P}$ be a D-labelled PDS such that for every rule $\langle p, \gamma \rangle \xhookrightarrow{d} \langle p', w \rangle$ we have $d \in D_{|w|}$. As a consequence, the GPP computation needs to deal exclusively with weights D_0 on transitions. It then suffices to prohibit infinite descending chains inside D_0 to guarantee termination and correctness of the GPP procedure. Let us call such a pair $\langle \mathcal{W}, \mathcal{S} \rangle$ a *partitioned* WPDS.

We come back to the question of how to encode programs with data. Given finite sets of global (G) and local (L) valuations, we suggested in Section 2.1.3 to encode globals in the control locations and locals in the stack alphabet; then a call rule has the form

$$\langle g, \langle n, \ell \rangle \rangle \hookrightarrow \langle g', \langle n', \ell' \rangle \langle n'', \ell'' \rangle \rangle,$$

where n, n', n'' describe the control flow, and $g, \ell, g', \ell', \ell''$ the global and local values involved in the step. Evidently, the number of such rules can quickly become huge even for modest sizes of G and L. A remedy is to represent data values *symbolically*, e.g., using *binary decision diagrams* or *BDDs* [Bry86]. In that setting, a PDS rule describes the control flow, e.g., $\langle \cdot, n \rangle \hookrightarrow \langle \cdot, n'n'' \rangle$, while all data tuples corresponding to the same flow of control are captured by a relation R, representable by a BDD. This representation is chosen, e.g., in older versions of Moped [Scha] or in Slam [BR00]. An alternative point of view is to see R as semiring weight in a WPDS.

Now, such relations have different arities. Let us denote $\mathit{Stk}[m, n] := G \times L^m \times G \times L^n$; an element of $\mathit{Stk}[m, n]$ describes the data part of two configurations that are of stack height m, n, respectively. Then, the relation for a call rule is of type $\mathit{Stk}[1, 2]$, for a pop rule it is $\mathit{Stk}[1, 0]$. For a configuration $c \in P \times \Gamma^n$, its weight $\delta(c)$ in the solution of a basic GPP problem should logically be of type $\mathit{Stk}[n, 0]$, corresponding to the intuition that the path from c to some $c' \in P \times \{\varepsilon\}$ is a concatenation of n pop sequences. This motivates the definition of a semiring whose domain D is the powerset of

$$\bigcup_{n,m \geq 0} \mathit{Stk}[m, n].$$

We describe the generalization of a relational 'join' operation that computes the net effect of two subsequent changes to the data configuration. Let us use $\ell_{[m,n]}$ as shorthand for $\ell_m, \ldots, \ell_n$. We call $e \in \mathit{Stk}[n, m]$, $e' \in \mathit{Stk}[k, p]$ *compatible* if they agree on $\min\{m, k\}$ positions 'in the middle', i.e. they are

of the form $e = \langle g, \ell_{[1,n]}\ ;\ g', \ell'_{[1,m]}\rangle$ and $e' = \langle g', \ell'_{[1,k]}\ ;\ g'', \ell''_{[1,p]}\rangle$. Then, for a compatible pair e, e', let

$$e \cdot e' := \begin{cases} \langle g, \ell_{[1,n]}\ ;\ g'', \ell''_{[1,p]}, \ell'_{[k+1,m]}\rangle & \text{if } m > k \\ \langle g, \ell_{[1,n]}\ ;\ g'', \ell''_{[1,p]}\rangle & \text{if } m = k \\ \langle g, \ell_{[1,n]}, \ell''_{[m+1,k]}\ ;\ g'', \ell''_{[1,p]}\rangle & \text{if } m < k \end{cases}$$

We can now describe the join operation on $d_1, d_2 \in D$ as follows:

$$d_1 \circ d_2 := \{\, e_1 \cdot e_2 \mid e_1 \in d_1 \ \wedge\ e_2 \in d_2 \ \wedge\ e_1, e_2 \text{ compatible} \,\}$$

Now, we can state the semiring to be used for BDD operations as $\mathcal{S} = \langle D, \cup, \circ, \emptyset, \mathrm{id}_G\rangle$, with $\mathrm{id}_G = \{\, (g; g) \mid g \in G \,\} \subseteq \mathit{Stk}[0, 0]$ as the neutral element of $\circ$. This semiring is unbounded because D is infinite. However, we obtain a partitioned WPDS if the rule weights respect the aforementioned encoding of data relations, with $D_i := \mathit{Stk}[1, i]$ for $i \geq 0$. For each configuration $c \in P \times \Gamma^n$ the GPP solution $\delta(c)$ is the product of n transition weights of type D_0, which is is of type $\mathit{Stk}[n, 0]$. Thus, the weight for c is can be seen as its complete data configuration consisting in the globals and n stack frames with locals.

We conclude that Algorithm 1 is applicable to this setting, and it is used in current versions of the model checker MOPED, which is based on the aforementioned WPDS library. Notice that for formal reasons, we presented the $\circ$ operation in such a way that operands may contain tuples of different arities. In practice, all relations that appear in the GPP computation are subsets of some $\mathit{Stk}[m, n]$, for some values of $m, n \leq 2$, and can hence be conveniently implemented as BDDs.

An alternative approach to integrating BDDs into the WPDS framework has been presented by Lal et al [LRB05] under the name *Extended WPDS*. These extend the WPDS framework with so-called *merging functions* that take care of saving/restoring locals during a call/return pair.

Differential GPP

The BDDs that appear in typical examples treated by MOPED can be quite large, and the GPP saturation procedure may be forced to update the weight of some transitions very often, i.e. case (ii) of the procedure, where d'' is replaced by $d_{new} := d' \oplus d''$. This can happen, e.g., if the underlying control flow of the WPDS contains a loop. Now, d_{new} may not differ from d'' by

‘much’, so it can be more efficient to propagate the difference between d_{new} and d'' rather than d_{new} itself. Section 5 of [RSJM05] provides a generic, *differential* version of the GPP algorithm that implements this idea (which is available in Moped).

2.2.2 Unbounded semirings

This section is based on common work with Morten Kühnrich, Jiří Srba, and Stefan Kiefer, published in [KSSK09a] (short version) and [KSSK09b] (including proofs). We study a relaxation of the boundedness condition on semirings, imposed in Section 2.2.1. The notion of *resource problems* is new and introduced here to clarify the contribution; this term was not used in [KSSK09a, KSSK09b].

In order to discuss this contribution and related work, it is first useful to recall that the GPP problem can be reduced to that of computing the pop triples, which – for the unweighted case – gives rise to a boolean-valued equation system, see (2.1). This idea can be extended to the weighted case.

As usual, fix a WPDS $\mathcal{W} = \langle \mathcal{P}, \mathcal{S} \rangle$, where $\mathcal{P} = \langle P, \Gamma, D, \Delta \rangle$ and $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1} \rangle$ is an idempotent semiring. Let $\mathcal{X} := \{ [\![p, \gamma, q]\!] \mid p, q \in P,\ \gamma \in \Gamma \}$ be a set of D-valued variables. We associate an equation with each variable:

$$\begin{aligned} [\![p,\gamma,q]\!] = & \bigoplus_{\langle p,\gamma\rangle \overset{d}{\hookrightarrow} \langle q,\varepsilon\rangle} d \quad \oplus \bigoplus_{\langle p,\gamma\rangle \overset{d}{\hookrightarrow} \langle r,\gamma'\rangle} (d \otimes [\![r,\gamma',q]\!]) \\ & \oplus \bigoplus_{\substack{\langle p,\gamma\rangle \overset{d}{\hookrightarrow} \langle r,\gamma'\gamma''\rangle \\ s \in P}} (d \otimes [\![r,\gamma',s]\!] \otimes [\![s,\gamma'',q]\!]) \end{aligned} \tag{2.2}$$

Equation (2.2) lists all possible ways how $\langle p, \gamma \rangle$ can be reduced to $\langle q, \varepsilon \rangle$. Intuitively, and in analogy to (2.1), its value summarizes the paths for that pop triple. So, when a greatest ($\sqsupseteq_{\mathcal{S}}$) solution of (2.2) exists and d is the value of $[\![p, \gamma, q]\!]$ in that solution, then we want d to have the value

$$\langle\!\langle p, \gamma, q \rangle\!\rangle := \bigoplus \{ \, val_{\mathcal{S}}(\sigma) \mid \langle p, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^{*} \langle q, \varepsilon \rangle \, \}.$$

When $\mathcal{S}$ is bounded, the desired solution can in principle be found by initially assigning $\bar{0}$ to all variables and then applying Kleene's fixpoint method (see below), which eventually terminates at the greatest fixpoint.

Algorithm 1 is a more efficient solution for this problem, which exploits the special structure of the equations.

When $\mathcal{S}$ is not bounded, the Kleene iteration may not terminate, and the same applies to Algorithm 1. A first alternative was given by Bouajjani, Esparza, and Touili, who studied the case of unbounded semirings, but where $\otimes$ is commutative [BET03a]. Thereafter, fixpoint equations over idempotent semirings gained more attention in the community: Inspired by work of Hopcroft and Kozen [HK99], Esparza, Kiefer, and Luttenberger developed algorithms based on Newton's method [EKL07b, EKL07a, EKL08], where the boundedness condition is replaced by *ω-continuity*, requiring that the infimum of every infinite set exists. Gawlitza and Seidl considered systems of equations over the integer semiring [GS07a] and rationals [GS07b], and Leroux and Sutre presented an algorithm for computing least fixed-points for so-called bounded-increasing functions over integers [LS07].

In [KSSK09a], we made another contribution to this area, motivated by some applications in dataflow analysis. These can be characterized as *resource problems*:

> Given a WPDS $\mathcal{W} = \langle \mathcal{P}, \mathcal{S}_i \rangle$ with integer semiring $\mathcal{S}_i := \langle \mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0 \rangle$ and $\mathcal{C} \subseteq \mathit{Conf}(\mathcal{P})$, find the maximal value $k \in \mathbb{Z}$ such that $\delta(c) \geq k$ for all configurations c. If no such bound exists, set $k := -\infty$.

In this setting, we can interpret weights as resources that are produced and consumed by the rules. If $k \geq 0$, then every path towards $\mathcal{C}$ produces all the resources it consumes. If $k < 0$, then there is a path that consumes more than it produces, which may point to a problem in the system under verification. However, this problem could in principle be fixed providing the system with $-k$ additional resources. On the other hand, if $k = -\infty$, then losses can be arbitrarily high, which may indicate serious resource leakage. Some concrete examples of resource problems are named in Section 2.2.3. A 'forward' variation of the problem is to ask whether the system, starting in a given configuration, has an execution that eventually runs out of resources. Since this is reducible to a 'backwards' problem, we do not discuss it explicitly (see [KSSK09b], Appendix C, however).

Example 2.12 *Let us consider examples with semiring $\mathcal{S}_i$. In all cases, we set $\mathcal{C} := \{\langle q, \varepsilon \rangle\}$.*

(i) *Take a WPDS $\mathcal{W}_1$ with $\mathcal{S}_i$ and three rules:*

$$\langle p, A\rangle \xhookrightarrow{2} \langle p, B\rangle \qquad \langle p, B\rangle \xhookrightarrow{-3} \langle p, C\rangle \qquad \langle p, C\rangle \xhookrightarrow{2} \langle q, \varepsilon\rangle$$

There is only one chain of rule applications from $\langle p, A\rangle$ to $\langle q, \varepsilon\rangle$, and the weight becomes negative in between, hence $k = -1$ in this case.

(ii) *For $\mathcal{W}_2$, consider the following rules:*

$$\langle p, A\rangle \xhookrightarrow{1} \langle p', \varepsilon\rangle \qquad \langle p', A\rangle \xhookrightarrow{-1} \langle p', \varepsilon\rangle \qquad \langle p', A\rangle \xhookrightarrow{1} \langle q, \varepsilon\rangle$$

Repeated application of the second rule can cause arbitrarily high resource leakage, hence $k = -\infty$.

(iii) *Let us consider a third example $\mathcal{W}_3$ with:*

$$\langle p, A\rangle \xhookrightarrow{1} \langle p, AB\rangle \qquad \langle p, A\rangle \xhookrightarrow{1} \langle q, B\rangle \qquad \langle q, B\rangle \xhookrightarrow{-2} \langle q, \varepsilon\rangle$$

Here, one part of the system produces an unbounded number of resources by pushing Bs onto the stack, but in the second phase, when removing the Bs, it consumes too many of them, so again $k = -\infty$.

The relation of resource problems to Kleene iteration is a bit subtle: $\mathcal{S}_i$ is not bounded, yet Kleene iteration would still terminate on $\mathcal{W}_1$ and $\mathcal{W}_2$ but not on $\mathcal{W}_3$. But k is finitely negative in $\mathcal{W}_1$ and $-\infty$ in both $\mathcal{W}_2$ and $\mathcal{W}_3$.

The main contribution of [KSSK09a] is an algorithm that detects when Kleene iteration is not going to terminate and reports this fact to the user, also indicating a reason for non-termination. If, however, Kleene iteration does happen to terminate, then our algorithm produces the greatest solution of the equation system. We identify a family of semirings where non-termination can be detected quite quickly.

To illustrate the contribution abstractly, one could say that the method in this section relates to the one from Section 2.2.1 like the Bellman-Ford shortest-path algorithm [Bel58, For56] relates to Dijkstra's algorithm [Dij59]. For this to work, we require two algebraic properties of a semiring $\mathcal{S}$: (i) $\sqsubseteq$ is a total order; (ii) $\mathcal{S}$ must be *inequality-preserving*.

Definition 2.9 *Let $\mathcal{S} = \langle D, \oplus, \otimes, \bar{0}, \bar{1}\rangle$ be an idempotent semiring. We call $\mathcal{S}$* inequality-preserving *if for all $a, b, c \in D \setminus \{\bar{0}\}$ we have that $a \neq b$ implies $a \otimes c \neq b \otimes c$. A totally ordered, inequality-preserving idempotent semiring is called a* tipi.

Examples of tipis include $\mathcal{S}_i$ from above or the semiring over the rationals between 0 and 1 with multiplication, $\mathcal{S}_r := \langle \mathbf{Q}[0,1], \max, *, 0, 1\rangle$.

Let us introduce some notation for the Kleene iteration. For every variable $x \in \mathcal{X}$, we let x_i denote its value in the i-th iteration, and denote by $\mathcal{X}_i$ the mapping of each variable x to x_i. Then for all $x \in \mathcal{X}$, we set $x_0 := \bar{0}$ and x_{i+1}, for $i \geq 0$, the value obtained by substituting all $x' \in \mathcal{X}$ by the value x'_i in the equation for x. We say that $\mathcal{X}_i$ is a fixpoint of (2.2) if $\mathcal{X}_i = \mathcal{X}_{i+1}$.

The following proposition summarizes results from [KSSK09a]:

Proposition 2.2 *Let $\mathcal{S}$ be a tipi. Then the Kleene iteration terminates after at most $n+1$ steps, or not at all, where $n = |\mathcal{X}|$, i.e.:*

(i) The Kleene iteration on (2.2) has a fixpoint iff $\mathcal{X}_n = \mathcal{X}_{n+1}$. If such a fixpoint exists, then $\mathcal{X}_n$ is the greatest solution of (2.2).

If the greatest solution exists, then additionally:

(ii) $[\![p,\gamma,q]\!]_n = \langle\!\langle p,\gamma,q\rangle\!\rangle$ holds for all $[\![p,\gamma,q]\!] \in \mathcal{X}$.

(iii) Let $\mathcal{C} := F \times \{\varepsilon\}$ for some $F \subseteq P$ be a basic set. Construct the $\mathcal{W}$-automaton $\mathcal{A}' = \langle P, \Gamma \times D, \rightarrow, P, F\rangle$ with $p \xrightarrow[d]{\gamma} q$ iff $[\![p,\gamma,q]\!]_n = d$. Then $\mathcal{A}'$ is the solution of the GPP for $\mathcal{W}, \mathcal{C}$ in the sense that it accepts each $c \in Conf(\mathcal{P})$ with weight $\delta(c)$.

(iv) $\mathcal{X}_n$ can be computed in $\mathcal{O}(|P|^3 \cdot |\Delta|^2)$ time (cf Remark 2.2).

Notes on the proof The proof is in multiple steps. First, in Section 2 we show that (i) holds for *any* polynomial equation system over tipis with n variables. This is based on the construction of witness graphs (called *derivation trees* in [KSSK09b]) along the same lines as in Section 2.2.1, with nodes $\langle x, d\rangle \in \mathcal{X} \times D$ explaining why variable x has value d. Suppose that $x_{n+1} \neq x_n$ for some $x \in X$. Then the witness tree for $\langle x, x_{n+1}\rangle$ has height $n+1$. The proof directly exploits the algebraic properties of tipis to argue that the witness tree contains a path from $\langle y, d_1\rangle$ via $\langle y, d_2\rangle$ to $\langle x, x_{n+1}\rangle$ for some $y \in X$ and $d_1 \sqsupset d_2$ (because of total order). Then the part of the witness graph between $\langle y, d_1\rangle$ and $\langle y, d_2\rangle$ can be 'pumped', which, due to inequality-preservation, yields ever smaller values for x.

In Section 3 of [KSSK09b] we argue that the fixpoint (if it exists) indeed gives the desired result and can be used to construct a solution for the GPP.

The complexity result comes from the fact that it only makes sense to have a variable $[\![p, \gamma, q]\!]$ if there exists at least one rule with $\langle p, \gamma \rangle$ on the left-hand side. Thus, the number of variables n is bounded by $|P| \cdot |\Delta|$ and that of the equation system by $|P|^2 \cdot |\Delta|$.

If the greatest fixpoint exists, then a witness graph can be constructed as in Section 2.2.1, while solving the equation system. If such a fixpoint does not exist, then there is at least one variable $x \in \mathcal{X}$ such that $x_n \neq x_{n+1}$. Let us call such a variable a *spoiler*.[3] These spoilers correspond to possible sources of problems if, e.g., $\mathcal{W}$ represents a resource problem as explained above, and reporting these spoilers to the user may help pinpoint the problems. In fact, it is possible to identify all spoilers and compute the precise values for all other variables, see [KSSK09a], Remark 1.

Example 2.13 *In the WPDS $\mathcal{W}_1$ and $\mathcal{W}_2$ from Example 2.12, the Kleene iteration will terminate and for each case produce an automaton $\mathcal{A}'$. To find the bound k, we are now interested in finding the minimal-weight path in $\mathcal{A}'$ leading from some arbitrary state to q (recall $F = \{q\}$). This can be done by applying the* ordinary *Bellman-Ford algorithm to $\mathcal{A}'$ interpreted as a graph. Doing so would yield $k = -1$ for $\mathcal{W}_1$ and $k = -\infty$ for $\mathcal{W}_2$.*

In the WPDS $\mathcal{W}_3$, the variable $\langle p, A, q \rangle$ will be identified as spoiler and we can directly conclude that $k = -\infty$.

The aforementioned work by Gawlitza and Seidl [GS07a, GS07b] is in some aspects closest to ours. It considers richer classes of equations but does not allow multiplication between variables, used in (2.2). Also, it directly exploits properties of the integers and rationals while our semirings rely on more general algebraic properties. Our generalization of Bellman-Ford was inspired by the one used in [GS07a].

2.2.3 Applications

The weighted PDS framework has a number of applications in verification and dataflow analysis. Starting with the latter, dataflow analysis is generally concerned with summarizing, for each node n in the control flow of a program, some aspect of the possible memory configurations that hold whenever control reaches n.

[3] Called witnesses in [KSSK09a], which has a different meaning in this document.

```
int x;                    void p()
                            if (...)
void main()                   return;
  x = 5;                    else if (...)
  p();                        x = x + 1;
  n3: return;                 n8: p();
                              x = x - 1;
                            else
                              x = x - 1;
                              n11: p();
                              x = x + 1;
                            fi
```

Figure 2.10: Example program for analysis of linear-constant propagation.

The seminal work of Sharir and Pnueli [SP81] on *interprocedural* dataflow analysis shows how to compute dataflow information capturing only the interprocedurally valid paths, i.e. those paths in which all return statements lead back to the site of the most recent call, by constructing an appropriate fixpoint equation system. However, [SP81] computes only one dataflow value for each program point, merging together all the paths that reach it, regardless of the calling context.

The WPDS framework that we presented in the previous sections provides new algorithms for interprocedural dataflow analysis. In particular, the WPDS framework allows to pose dataflow queries with respect to a regular language of stack configurations.

Example 2.14 *Consider the program in Figure 2.10, which has one global integer variable x. Without making any assumptions on the range of x, we show in [RSJM05], Section 4.3, how to frame the problem of linear-constant propagation as a bounded WPDS problem. This allows not only to prove that the program always terminates with $x = 5$, regardless of the recursion depth in procedure* `p`*. A WPDS reachability query also allows to determine, for instance, that $x = 5$ always holds whenever one enters* `p` *with alternating calls from both recursive sites, i.e. whenever the stack is a word in the regular language $(n_8 n_{11})^* n_3$.*

Conventional interprocedural dataflow-analysis algorithms, by merging

together all calling contexts, would only be able to provide the answer that the value of x when entering `p` can be variable. In [RSJM05], Section 4.2, we give an algorithm that allows to extract this conventional data-flow information from the $\mathcal{W}$-automata produced by Algorithm 1. Thus, the WPDS framework is strictly more general than conventional data-flow analysis. Another contribution over conventional data-flow analysis is the generation of interprocedural witness paths, which was not considered previously. A precise and extensive overview of the relation to pre-existing dataflow analysis procedures is given in [RSJM05], Section 6.

Applications for this exist, for instance, in *program understanding*, where users can pose queries about dataflow information with respect to a regular language of initial stack configurations and/or demand an explanation (in the form of witness paths) for the values provided by the analysis tool. Also, *program optimizers* could make queries about dataflow values in different calling contexts, allowing them to produce different versions of a procedure that can be used and optimized separately according to context.

Bounded idempotent semirings can model a variety of standard dataflow analyses, such as the so-called *bitvector problems*, e.g. live-variable analysis.[4] The applicability of WPDS to other dataflow analyses such as constant propagation and linear-constant propagation were demonstrated by us in [RSJM05]. Müller-Olm and Seidl [MOS04] provided an interprocedural version of affine-relation analysis, which determines, for each program point n, the set of all affine relations that hold among program variables whenever n is executed. This method can be alternatively be framed as a GPP problem and is now used in the CODESURFER tool for x86 executables [BGRT05].

Resource problems have been identified as a type of application for the unbounded case in Section 2.2.2. Some examples of such problems are listed in [KSSK09a], Section 4. These include memory allocations in the Linux kernel, where the goal is to prevent memory corruption. Another are correspondance assertions [WL93] used in the analysis of authentication protocols. Here, correct usage of a protocol is specified by annotating programs with additional labels that form pairs of `begin` and `end` statements, which may be inserted anywhere in the code, i.e. their location in the code may be unrelated to its syntactic structure. The sequence of `begin` and `end` statements along a correct run must be well-formed. Thus, a `begin` statement

[4] Bitvector problems in a PDS context were previously adressed by Esparza and Knoop [EK99] using specialized *pre** and *post** queries.

corresponds to creation of a resource and `end` to its consumption. A third application is to test shape-balancedness in context-free language, e.g. to test whether the XML documents generated by some program are always well-formed. Here, our framework give an $\mathcal{O}(n^4)$ algorithm, whereas for a previous solution [TM07] no complexity bound was known.

In verification, the pushdown-based model checker Moped uses the WPDS framework to combine PDS with binary-decision diagrams (BDDs); details of this were discussed in Section 2.2.1. The semiring $\mathcal{S}_1$ from Example 2.7 can be used to produce shortest counterexamples or reachability witnesses.

Other uses in verification that have been found for the WPDS framework are in the analysis of *concurrent PDS*, where two PDS $\mathcal{P}_1, \mathcal{P}_2$ synchronize via common actions. Given configurations $c_1 \in \mathit{Conf}(\mathcal{P}_1)$ and $c_2 \in \mathit{Conf}(\mathcal{P}_2)$, it is undecidable whether c_1, c_2 can be jointly reached at the same time, by reduction from the problem of testing emptiness of the intersection of two context-free languages [Ram00]. However, $\delta(c_1)$ and $\delta(c_2)$ can be seen as *abstractions* or *approximations* of the context-free languages of synchronisation actions leading to c_1 and c_2 and can therefore yield a 'safe' answer for joint reachability that only errs on one side. This application was initially proposed by Bouajjani, Esparza, and Touili in [BET03b, BET03a]. While their approach was formulated in a different framework and under slightly different algebraic restrictions on semirings, all but one of the abstractions proposed there could be formulated in the bounded WPDS framework, providing a more efficient solution than in [BET03b]; this solution was subsequently integrated into [BET03a]. Later, Touili and others expanded this idea into an abstraction-refinement mechanism inside the model checker Magic [CCK+06].

Finally, Wenner studied an extension of the weighted framework to *dynamic pushdown networks*, using this to solve dataflow problems for a restricted class of concurrent PDS [Wen10].

2.3 Other PDS-related contributions

This chapter summarizes some other contributions related to pushdown systems. These concern the development of tools (Section 2.3.1), research on concurrent PDS (Section 2.3.2), an abstraction-refinement scheme (Section 2.3.3), and the use of PDS in authorization (Section 2.3.4).

2.3.1 Tool development

Tools are presented first in this section, since they provided the backbone for the other contributions. The saturation and witness-generation procedures for the bounded case shown in Section 2.2.1, along with other related functionality, have been implemented in two libraries called WPDS [Schc]. To instantiate the framework, all a user has to provide are application-dependent functions that implement the operations $\oplus$ and $\otimes$, which will be appropriately invoked by the libraries. An alternative implementation, made by Reps' group at Wisconsin is called WPDS++ resp. WaLi [KRML06].

Based on this library, I re-write the tool Moped from the ground up, which enabled me to give it cleaner, more versatile and structured architecture that other developers could build upon. For instance, Stefan Kiefer, then a master student in our group, used this framework to integrate an extension for counterexample-based abstraction refinement (see Section 2.3.3).

Moped then matured into a user-friendly tool for analyzing Java programs (jMoped), mostly through the work of Dejvuth Suwimonteerabuth, a PhD student in our group at the time. We turned jMoped into a test environment for Java programs [SSE05, SBSE07]. The idea of jMoped is to translate Java bytecode into a PDS, automatically including the the bytecode of available libraries if need be. Felix Berger contributed an Eclipse plugin, which Dejvuth later expanded; more details can be found in his thesis [Suw09].

2.3.2 Verification of concurrent PDS

Pushdown systems as such were conceived as entirely sequential models of computation with no thought of concurrency at all. However, it is natural to make the model (and the aforementioned results) applicable also in a concurrent and distributed setting, and this subject has attracted considerable attention by the verification community in the last decade.

Verification of concurrent PDS is challenging not only due to state-space explosion, but also due to the decidability boundary. For instance, a system with two stacks and one common variable is equivalent to a Turing machine.

A very influential approach has been the proposal of *context-bounded analysis* by Qadeer and Rehof [QR05], where one only considers executions in which the active process changes no more than a fixed number of times. My own contributions in this area pertain to that approach and span two publications [BESS05, SES08].

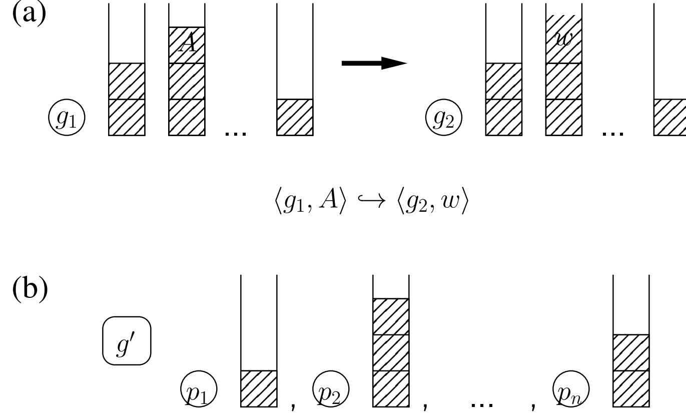

Figure 2.11: (a) CPS executing a transition in thread 2; (b) configuration of an APN.

Context-bounded reachability analysis The model proposed in [QR05] is called *concurrent pushdown systems* (CPS). Configurations of a CPS consist of one control location (from a set G) and multiple, say n, stacks over alphabet Γ. Every thread operates like a PDS, it can query and manipulate the common control location and one of the n stacks. Figure 2.11 (a) illustrates the idea, where thread number 2 makes a step, changing both control location and its own stack.

A *context* is a sequence of transitions working on the same stack, i.e. by the same thread, and a sequence is k-bounded if it is a concatenation of k contexts. A k-context-bounded reachability analysis consists in computing the set of all forwards/backwards reachable states (from some regular set of initial/final configurations). Qadeer and Rehof justify this idea by citing experience [QW04] that most concurrency bugs can be found within few contexts.

Suppose that $n = 2$ and we start from control location $g_0 \in G$ and regular languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Gamma^*$ for the two threads. The reachability algorithm proposed in [QR05] is as follows: Starting from the tuple $\langle k, g_0, \mathcal{L}_1, \mathcal{L}_2\rangle$, one computes $\mathcal{L}' = post^*(\{g_0\} \times \mathcal{L}_1)$ (same for $\mathcal{L}_2$) using standard PDS reachability [BEM97, FWW97, EHRS00]. This corresponds to one context in the first thread. So one splits $\mathcal{L}'$ according to control location, i.e. into $|P|$ subproblems $\langle k-1, g, \mathcal{L}' \cap (\{g\} \times \Gamma^*), \mathcal{L}_2\rangle$, for each $g \in G$, and proceeds recursively until k contexts have been reached. Intuitively, the recursive computation resembles a tree, where different branches correspond to different sequences

of active processes *and* to different values of G during context switches. This procedure takes $\mathcal{O}(k^3 \cdot n^k \cdot |G|^{k+5})$ time, when all other, minor parameters are fixed.

A generalization of CPS This paragraph briefly explains the contribution of [BESS05], due to common work with Ahmed Bouajjani, Javier Esparza, and Jan Strejček.

In [BESS05] we provided some improvements over CPS in terms of complexity bounds and expressive power. In the model that we call APN (asynchronous pushdown network), we replace G by two sets of *global* and *thread-local* control states (G' and P, respectively). This allows to limit G' to the amount really necessary to exchange information between threads, whereas P can be used, e.g., to model a thread-internal return value in a pop rule. APN are therefore more concise. Figure 2.11 (b) illustrates the structure of a configuration of an APN with global control state g' and thread-local control states $p_1, \ldots, p_n$.

A context in an APN is a sequence where the same thread operates on the global control state, interspersed by local actions of other threads. For this notion of context we give a forwards reachability algorithm in $\mathcal{O}(k^2 \cdot n^k \cdot |G|^{k+2})$ time. (Notice that k is supposed to be small, so the improvement from $k+2$ to $k+5$ in the exponent matters.)

Moreover, we then consider an extension of the model with unbounded thread creation (bounded thread creation was allowed in [QR05]). Here, it turns out that the set of *forward* reachable configurations is context-free, whereas *backward* reachability preserves regularity. We provide algorithms for both cases.

Symbolic context-bounded analysis While the context-bounded approach generated some interest in the verification community (see related work below), actual implementations of the approach were not forthcoming at first. Its applicability was hampered by the difficulty to combine it with symbolic techniques, e.g. BDDs: After every individual *post** computation, the result is split into $|G|$ parts that are processed individually – exactly the opposite of what one intends to do in symbolic model checking.

In joint work with Dejvuth Suwimonteerabuth and Javier Esparza [SES08], we tackled this problem and developed a variant suitable for BDD-based methods. The main contribution of that paper is the development of *lazy*

splitting. Lazy splitting determines subsets of G that one can continue to treat in the same branch of the tree (see above), thereby limiting its width; the subsets can themselves be determined by BDD operations.

The algorithm has been implemented in jMoped, and we report on experiments with Java programs, such as the `java.util.Vector` class from the Java library and a Bluetooth driver.

Related work Before and since the publication of the works mentioned above, there have been many more works on verifying concurrent PDS, and in particular on context bounding. Some pointers to these are given below, but the list is necessarily incomplete.

Since reachability is undecidable for concurrent PDS in general, researchers have tried to either find interesting, restricted subclasses that still allow at least reachability analysis, or provide approximative solutions for more general cases.

In the first category, the proposals include restricting the communication architecture [BMOT05, ABT08, Wen10], communication via locks used either in nested fashion [KIG05] or in lock chains [Kah09], or more recently asynchronous communication via FIFO channels [HLMS10] with various restrictions, e.g. a process may send messages only when its stack is empty.

An example of over-approximative solutions is [BET03a], already discussed in Section 2.2.3, which computes an abstraction of the language of synchronization actions in each parallel component. The context-bounded approach belongs into the class of under-approximations and has been among the most influential ideas, having spawned several variations and extensions, e.g. analysis of heap structures [BFQ07] and queue systems [LMP08], bounding only the number of context switches between a push and corresponding pop [LN11], or fixing other parameters like the number of times a node can switch between sending and receiving [BE12].

In parallel with our work on BDD-based context-bounded analysis, Lal et al [LTKR08] published another approach based on weighted transducers. Moreover, some work has gone towards reducing concurrent reachability to sequential reachability [QW04, LR09, LMP09].

2.3.3 Abstraction-refinement for symbolic PDS

This section gives a brief account of work done together with Javier Esparza and Stefan Kiefer [EKS06, EKS08b]. We studied counterexample-guided

abstraction refinement (CEGAR) in the context of a symbolic pushdown model checker like Moped.

CEGAR is a powerful, generic methodology for verification of systems with large or infinite state spaces, introduced by Clarke et al [CGJ+00]. In principle, the goal is to check whether a certain target state s is reachable in some LTS $\mathcal{T}$. If the state space is very large, one can subdivide the states into equivalence classes and obtain an LTS $\mathcal{T}_\equiv$ over their quotient set through existential abstraction, then analyze $\mathcal{T}_\equiv$ instead. Since $\mathcal{T}_\equiv$ is an overapproximation, this can yield a 'spurious' counterexample, e.g. a path to s that is not possible in $\mathcal{T}$. This path is then used to refine the equivalence classes, and the process repeats, until either the target state s is shown to be unreachable or a real counterexample emerges.

In [EKS06, EKS08b] we studied this approach in the context of pushdown model checking where relations are encoded as weights in a WPDS (see Section 2.2.1). The input language of Moped, called Remopla, allows to specify a program with procedures, basic data types like integers and booleans, and, e.g., arrays over those types. Statements, such as guards or assignments, are translated into BDDs in standard fashion. We then wish to apply CEGAR in order to reduce the amount of work that the model checker needs to do. In the parlance of the explanation above, our LTS $\mathcal{T}$ is represented by a weighted PDS $\mathcal{W}$ with BDDs, and $\mathcal{T}_\equiv$ by another such weighted PDS $\mathcal{W}_\equiv$ with *smaller* BDDs.

A nice feature of weighted PDS in this context is the witness graph discussed in Section 2.2.1. Recall that this witness graph allows to explain the weight $\delta(c)$ of a given configuration c in $\mathcal{W}_\equiv$. If c represents the target configuration, then $\delta(c)$ contains the data configurations that can hold at c, and the witness graph provides a path in the PDS for all those data configurations. We simulate those paths to check whether at least one of them corresponds to a path in $\mathcal{W}$. If that is the case, we can terminate and say that the target configuration is reachable. Otherwise, the witness graph is 'spurious', and we need to refine the abstraction. Thanks to the structure of the graph, we can obtain *all* spurious counterexamples in the current abstraction and refine the equivalence relation to eliminate them all in the next iteration, including in the case of recursive procedure calls.

Figure 2.12 shows a simple example (without procedure calls) where we want to know if the error is reachable. If, in the initial abstraction, all data is discarded, then the witness graph shows two counterexamples, one that passes through the loop once, and one that skips it. This witness graph is

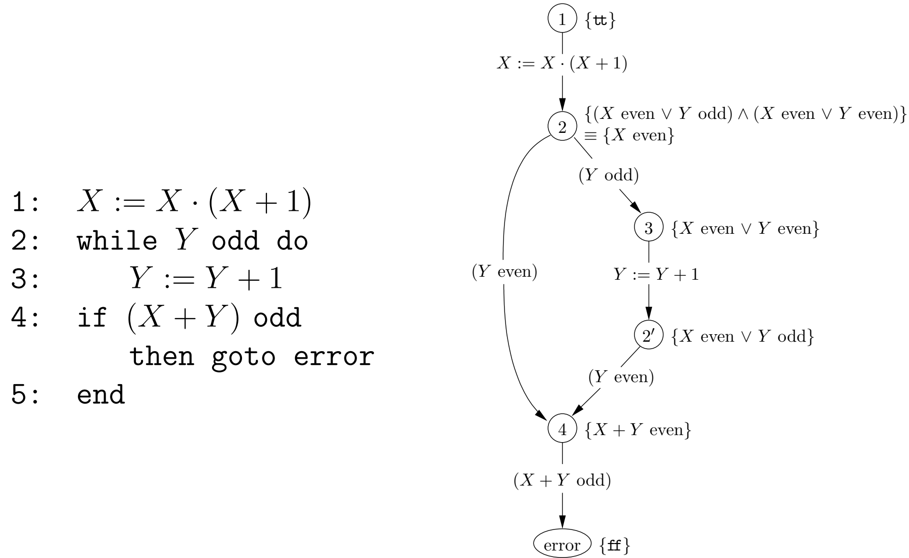

Figure 2.12: Program and witness graph annotated with weakest interpolants.

spurious, in fact the error is unreachable. The corresponding witness graph is shown on the right-hand side of Figure 2.12. Using standard BDD operations, one can compute, for each node n, either the strongest postcondition I_n (i.e. the facts that are guaranteed to hold after executing the paths leading to that node), or the weakest precondition J_n for making the rest of the trace after n infeasible. Figure 2.12 shows the weakest preconditions as annotations next to each node.

In [McM03], McMillan proposed to use Craig interpolation to automatize abstraction refinement. Given a spurious counterexample path, Craig interpolation provides, at each point n along the path, a predicate I that talks only about the variables at point n. Such a predicate is called an interpolant, and it will be added to the refinement of $\mathcal{T}_{\equiv}$. If chosen carefully, this addition guarantees that the spurious path is ruled out in the future.

[McM03] uses SAT checkers, and the interpolant for a pair $F \wedge G$ is obtained from a resolution proof for insatisfiability of $F \wedge G$. In [EKS08b], we use BDDs instead. We show that the two families of predicates (strongest and weakest, I_n and J_n) are both suitable for ruling out spurious paths. Moreover, we show that the computation of Craig interpolants works well with BDDs. We also introduce another kind of interpolants, called *conciliated*, which lie

between the strongest and weakest interpolant but are simpler in the sense that they talk about fewer variables.

Based on these two components – exploitation of witness graphs and Craig interpolants – the approach was integrated into Moped, and [EKS08b] provides a number of detailed case studies, together with several case studies.

Related work CEGAR has been used in several other software model checkers, such as Slam [BR01], Blast [HJMS02], or Magic [CCG+03]. The approach in [EKS08a] differs from these by the consistent use of BDDs. For instance, the Slam model checker [BR01] uses BDDs to represent the abstraction ($\mathcal{T}_\equiv$) but theorem proving to obtain the predicates. Magic does not use BDDs at all, but relies on SAT solvers and theorem provers.

2.3.4 Authorization systems

The main issues in access control of shared computing resources are *authentication*, *authorization* and *enforcement*. The first aspect concerns the identification of principals. The second aspect demands the formulation of a policy detailing which access rights should be granted to each principal. Enforcement addresses the problem of implementing the authorization during an execution. Here we discuss the second aspect.

To explain the necessity of a sophisticated authorization system with clear formal semantics, let us start with a concrete example: The computer science department at ENS Cachan runs a server ("*serveur pédagogique*") used by teachers and students to manage administrative aspects, marks for courses, ECTS credits, and so on. Every course belongs to a certain diploma (L3, M1, or M2). Teachers and students belong to various different establishments scattered over Paris and surroundings, who are jointly running a master programme.

Naturally, not all users in this system have the same rights, e.g. a student may not change his own marks. Thus, it becomes necessary to specify which users may exercise which functionalities on certain objects in the system. The initial system for granting authorization that was implemented on the server assigned to each user a set of rôles; he could be teacher, student, director, admin, etc. This worked to some extent, but it was not readily possible to require that, e.g., a teacher can change only the marks in his own course. Later, a second, more flexible system for managing access rights was installed, which permitted, e.g. to relate the access rights of a teacher or a student to

his own courses. However, the system knew only a few such relations, which were hardcoded. Also, the precise semantics of this authorization system was neither documented nor evident in all cases.

As so often, formal methods came to the rescue. In fact many interesting features for specifying access rights can be readily expressed using pushdown systems and variations thereof.

For instance, the director in charge of some diploma should be able to change certain settings in the courses belonging to that diploma. Suppose that course C belongs to the M2, and Alice is the director of the M2. In this simple example, to determine whether Alice has access to C, one could compute the join between two relations (cours-diploma and diploma-director) and check whether that contains the pair $\langle C, \mathit{Alice} \rangle$.

Let us regard another modelling, based on pushdown systems:

- the *authorization certificate* becomes a 'push' rule:

$$\langle C,\ \mathit{access} \rangle \rightarrow \langle C,\ \mathit{diploma\ director} \rangle;$$

- the *relation certificates* become 'pop' rules:

$$\langle C,\ \mathit{diploma} \rangle \rightarrow \langle M2, \varepsilon \rangle$$

  and

$$\langle M2,\ \mathit{director} \rangle \rightarrow \langle \mathit{Alice}, \varepsilon \rangle.$$

Then, checking whether Alice has access to C is equivalent to checking whether in the resulting PDS, we have $\langle C,\ \mathit{access} \rangle \rightarrow^* \langle \mathit{Alice}, \varepsilon \rangle$.

We see therefore that we have two kinds of data or 'certificates' that govern access rules: *relations* describe facts about the world, e.g. who is teacher of which course, who is director of which lab or manager of a certain diploma; or which course belongs to which diploma. These relations can be referred to and joined together using *authorizations*. An authorization grants a certain functionality (above: *access*) over a certain object (above: course C). It is possible to attach multiple functionalities to the same object, e.g., a restricted "*view*" of the course page for the students, or "*modify*" the settings of the course, the exam etc, which would be accessible only for the teachers. The words *view* and *modify* would simply replace *access* in the example above. In all cases, the authorization problem reduces to querying a pop triple in an underlying PDS. Moreover, the witness graph produces a

chain of certificates leading from the resource to the user and constitutes a valid proof for access. This framework has a clear, simple semantics and is yet very flexible; new functionalities can simply be added as a new keyword. New relations can also be added in a modular way.

Moreover, once one exploits the full expressiveness of PDS, one obtains more flexibility in specifying access rights, surpassing simple joins. Indeed, the SPKI/SDSI standard for expressing authorizations [EFL+99] (published as RFC 2693), is as expressive as pushdown systems. Hence, reachability algorithms for PDS solve the authorization problem for SPKI/SDSI (or for the *serveur pédagogique*). Let us note that SPKI/SDSI was designed for open-world scenarios, where public-key cryptography is used to sign certificates. Indeed, a principal is identified with his public key in certificates, and a principal's certificates are signed using his private key. In this way, the veracity of a certificate chain can be checked by any person in the middle, who may not even know the identitities of the persons involved. It suffices to execute the cryptographic primitives, using the key that is part of the certificate to verify that its signature is genuine.

It was initially observed by Jha and Reps [JR02] that pushdown systems are a suitable formal model for expressing authorizations in the SPKI/SDSI framework. We expanded on this thought by studying several extensions of pushdown systems and their counterparts in authorization. The work in this area was done with many people: Ahmed Bouajjani, Javier Esparza, Somesh Jha, Thomas Reps, Stuart Stubblebine, Dejvuth Suwimonteerabuth, and Hu Wang.

Weighted PDS [SJRS03] The framework of bounded idempotent semirings allows to annotate certificates with additional information, for instance concerning privacy, recency, validity, or trust. In this way, one can compute whether a user has an access right while minimizing the amount of sensitive information given away in the process.

Intersection certificates [SSE06] Alternating PDS allow to introduce "intersection certificates": a user may obtain an access if he belongs to the intersection of two groups. Indeed this possibility was also mandated for in RFC 2693. We show that the corresponding reachability problem is exponential in general but remains polynomial if intersection is restricted to authorization certificates.

Practical Steps [JSWR06, WJR$^+$06] To improve the practicability of the approach, we show how to implement SPKI/SDSI authorization in a distributed setting, i.e., where certificates are distributed over various servers. We discuss how to carry out the pre^* or $post^*$ procedure in a distributed way in this case. Another aspect concerns the integration into Kerberos.

Reputation systems [BESS08] In a related vein, we use the weighted pushdown framework with probabilities. We show how this can be used to compute the reputation of individuals within a group, for instance in academia, where individuals can distribute their trust to various conferences, journals, etc, and the trust allocated to these entities is in turn distributed to the authors who publish in those venues. For instance, we used the system to compute the approximative reputation of the PC members of the conference this paper was published in.

Finally, a subset of this functionality is nowadays being used to handle access control inside the “serveur pédagogique”. The programming for this implementation was carried out together with Paul Gastin and Audrey Halbert.

Chapter 3

Petri nets

Concurrency poses particular problems for the design and verification of systems. In Section 2.3 we mentioned the combination of pushdown techniques with concurrency. The combination of these two components poses a serious challenge for designing formal methods that are at the same time decidable, efficient, *and* permit to model meaningful classes of systems.

However, concurrent systems are challenging to design and analyze even without additional sources of complication. This holds true especially when a system consists of many components running at different, often impredictable speeds. This may result in astronomically many different orderings of execution, making the behaviour difficult to predict and bugs difficult to find or reproduce. Some of these orderings are important to understand the behaviour of a system. Let us imagine n components in parallel sharing a variable p, where the first action of process i (for $i = 1, \dots, n$) consists in setting $p := i$. It seems clearly important to regard the different orders of execution, or – depending on context – at least remember which process modified p last. On the other hand, let us imagine n processes, each with one action that is entirely independent of the other processes. In this case, the order of execution seems unimportant, and we would not be inclined to explore all 2^n interleavings.

The challenge, then, in automated verification of concurrent systems, is to design methods that can distinguish between interesting and uninteresting interactions. In this chapter, we base our endeavours on Petri nets. Originally introduced by Carl Adam Petri [Pet62] in his PhD thesis, they are a natural way to model concurrent systems, in which dependencies and causalities between components and actions are directly visible in the structure of

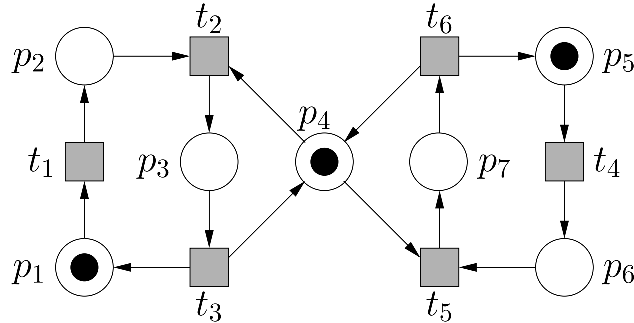

Figure 3.1: Petri net modelling a simple mutex protocol.

the model. Moreover, Petri nets have an appealing graphical representation. Many different variants of Petri nets have been proposed in the literature; here, we use simple place/transition nets. They consist of two type of nodes called *places* (usually round) and *transitions* (usually square). Places can carry *tokens*, whose distribution over the places, called *marking*, define the current state, symbolizing, e.g., control flow, variables, resources, etc. Transitions represent actions, and tokens can flow along the arcs, which connect places and transitions, symbolizing the consumption and/or production of resources. We point to Reisig's book [Rei98] for a survey on modelling and analysis methods for Petri nets.

Example 3.1 *Figure 3.1 shows a small Petri net modelling two processes competing to enter a critical section (places p_3, p_7), where p_4 takes the role of a semaphore protecting the access.*

In Richard Mayr's hierarchy of prefix-rewrite systems [May98], Petri nets are the parallel counterpart to PDS: while PDS replace one sequence in the prefix of a term by another sequence, Petri nets replace one parallel expression by another. But the two classes are far from being duals, concurrent systems have fundamentally different properties, and hence their verification faces completely different algorithmic challenges. This holds even when we regard the particular subclass of *bounded* nets, where no reachable marking contains a place occupied by more than k tokens, for some k (for $k = 1$, one speaks of *safe* nets).

The techniques in this chapter are based on *unfoldings*. An unfolding $\mathcal{U}$ of a net $\mathcal{N}$ is obtained, roughly speaking, by unrolling its loops. Thus, in

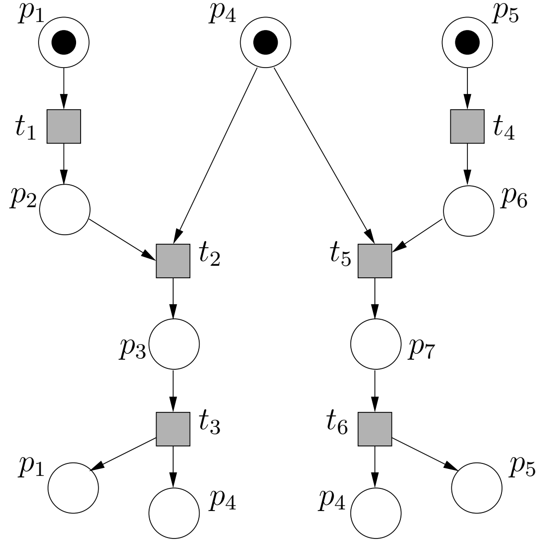

Figure 3.2: Initial part of the unfolding of the net in Figure 3.1.

general, $\mathcal{U}$ is infinite. Figure 3.2 shows the initial part of the unfolding for the net in Figure 3.1. Unfoldings were initially an object of semantic interest [NPW81], as the concurrent counterpart of a *computation tree*. McMillan [McM92] showed that they could be interesting for verification purposes because for a *bounded* net $\mathcal{N}$, a finite prefix $\mathcal{P}$ of $\mathcal{U}$ suffices to recover all reachable markings. This is interesting because $\mathcal{U}$ and $\mathcal{P}$ are structurally acyclic, so reachability is NP-complete for $\mathcal{P}$ but PSPACE-complete for $\mathcal{N}$. McMillan's result was followed up by Esparza et al [ERV02], showing that one can always obtain a prefix whose size is bounded by the number of reachable markings of $\mathcal{N}$, and is usually much smaller. Therefore, unfoldings may serve as a basis for further analyses. There is a large body of work describing their construction, their properties, and their use in various fields; see [EH08] for an extensive survey.

Example 3.2 *Figure 3.2 shows an unfolding prefix $\mathcal{P}$ of the net $\mathcal{N}$ from Figure 3.1. Note that places and transitions of $\mathcal{P}$ are* labelled *with places and transitions of $\mathcal{N}$, and Figure 3.2 shows these labels.*

The development of algorithms and tools [Schb, Kho] for unfoldings has mostly concentrated on bounded Petri nets, and we follow this tradition. Thus, we are not concerned with finding decidable yet expressive enough

subclasses of Petri nets, but concentrate instead on algorithmic aspect as well as efficient data structures and algorithms.

Before going to the technical details of the contributions, we first introduce basic notations and facts about Petri nets and unfoldings in Section 3.1. The line of work that I chose to highlight in this chapter is presented in Section 3.2 and concerns the attempts to make unfoldings more concise and hence efficient to use. Other contributions related to Petri nets are summarized in Section 3.3.

3.1 Basics of c-nets and unfoldings

The material presented in this section covers introductory material about nets and their unfoldings. In order to make this material applicable to the entire chapter, the introduction is made in terms of c-nets, which are more general than Petri nets, following [MR95, BCM98, BCM01]. Notions that are more specific to c-nets or the special case of Petri nets are deferred until Section 3.2 and Section 3.3, respectively. Section 3.2 also provides more background and motivation for studying c-nets.

3.1.1 Contextual nets

A contextual net is a Petri net extended with read arcs, which allows transitions to check for tokens without consuming them.

Definition 3.1 *A* contextual net (c-net) *is a tuple* $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$*, where* P *and* T *are disjoint sets of* places *and* transitions*,* $F \subseteq (P \times T) \cup (T \times P)$ *is the* flow relation*,* $C \subseteq P \times T$ *is the* context relation*, and* $m_0 \subseteq P$ *is the* initial marking*. A pair* $\langle p, t \rangle \in C$ *is called* read arc*.* $\mathcal{N}$ *is called* finite *when the* P *and* T *are finite. A* Petri net *is a c-net without read arcs.*

A marking *of* $\mathcal{N}$ *is a function* $m \colon P \to \mathbb{N}$*. A set* $m \subseteq P$ *(such as* m_0*) is equivalently treated as the marking where for all* $p \in P$ *we set* $m(p) = 1$ *if* $p \in m$ *and* $m(p) = 0$ *otherwise.*

Example 3.3 *Figure 3.3 (a) shows the usual graphical representation of a c-net. Read arcs are depicted as undirected arcs, such as* $\langle p, b \rangle$ *and* $\langle p, c \rangle$*. We say that* b *and* c read *from* p*, whereas* a produces *a token on* p *and* d *consumes one.*

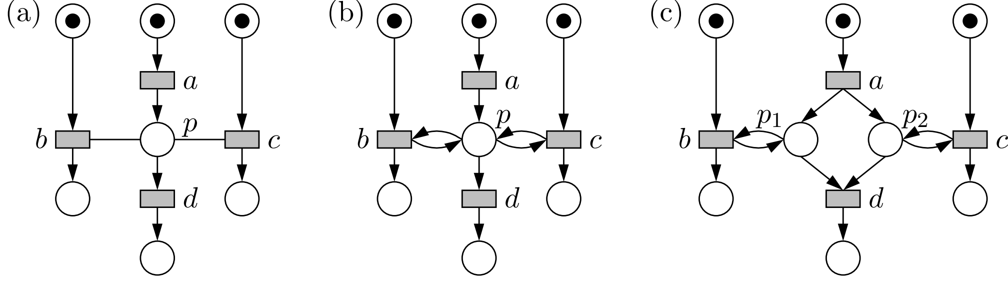

Figure 3.3: (a) C-net $\mathcal{N}$; (b) its plain encoding; (c) its PR-encoding.

For $x \in P \cup T$, we call ${}^\bullet x := \{\, y \in P \cup T \mid (y, x) \in F \,\}$ the *preset* of x and $x^\bullet := \{\, y \in P \cup T \mid (x, y) \in F \,\}$ the *postset* of x. The *context* of a place p is defined as $\underline{p} := \{\, t \in T \mid (p, t) \in C \,\}$, and the context of a transition t as $\underline{t} := \{\, p \in P \mid \langle p, t \rangle \in C \,\}$. These notions are extended to sets as usual. For the sake of simplicity, we assume for any transition t that its context is disjoint from its preset and its postset, i.e. ${}^\bullet t \cap \underline{t} = \emptyset$ and $t^\bullet \cap \underline{t} = \emptyset$.

A set $A \subseteq T$ of transitions is *enabled* at marking m if for all $p \in P$,

$$m(p) \geq |p^\bullet \cap A| + \begin{cases} 1 & \text{if } \underline{p} \cap A \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Then A can *occur* (or *fire*/*be executed*), leading to marking m', where $m'(p) = m(p) - |p^\bullet \cap A| + |{}^\bullet p \cap A|$ for all $p \in P$. We call $\langle m, A, m' \rangle$ a *step* of $\mathcal{N}$. When A is a singleton $\{t\}$, we simply say that t is enabled/fires etc.

Intuitively, the notion of enabledness requires that for every place p, m holds one token for each transition $t \in A$ that consumes p, plus an additional token if at least one transition reads from p. This means that a token cannot be read and consumed at the same time, and our notion of step does not require that actions in A happen completely synchronously. Contextual nets with this notion of enabling were initially introduced by Montanari and Rossi in [MR95]. A different notion of enabling that allows reading and consuming a token simultaneously is used, e.g., by Janicky und Koutny in [JK91].

Example 3.4 *Consider the c-net in Figure 3.4. Under our notion of step, the initial marking enables either t_1 or t_2 but not $\{t_1, t_2\}$. One transition disables the other, so t_3 can never fire.*

Denoting the set of markings by $\mathbb{N}^P$, we can associate with $\mathcal{N}$ an LTS (see Section 2.1) $\mathcal{T}^s_\mathcal{N} = \langle \mathbb{N}^P, 2^T, \rightarrow_s \rangle$, where $m \xrightarrow{A}_s m'$ if $\langle m, A, m' \rangle$ is a step

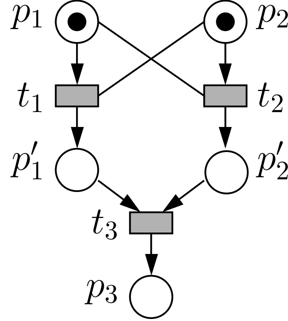

Figure 3.4: A net with a contextual cycle.

of $\mathcal{N}$, and another LTS $\mathcal{T}^i_{\mathcal{N}} = \langle \mathbb{N}^P, T, \rightarrow_i \rangle$, where $m \xrightarrow{t}_i m'$ if $\langle m, \{t\}, m' \rangle$ is a step of $\mathcal{N}$. The LTS $\mathcal{T}^s_{\mathcal{N}}$ is the *step semantics* of $\mathcal{N}$, whereas $\mathcal{T}^i_{\mathcal{N}}$ represents its *interleaving semantics*, also called the *reachability graph*.

A marking m is said to be *reachable* in $\mathcal{N}$ if $m_0 \xrightarrow{\sigma}^*_i m$ for some $\sigma \in T^*$. A marking m is n*-safe* if $m(p) \leq n$ for all $p \in P$. A c-net N is said to be n-safe if every reachable marking of N is n-safe. It is called *bounded* if there exists an n such that N is n-safe. A 1-safe net is simply called *safe*, and in this case we treat markings as sets.

3.1.2 Encodings

We present some encodings from c-nets into Petri nets that preserve their interleaving semantics or even their step semantics. Consider again the c-net $\mathcal{N}$ in Figure 3.3 (a). Place p has two transitions b, c in its context, modelling a situation where, e.g., two processes have read-only access to a common resource p. The step $\{b, c\}$ can occur in $\mathcal{N}$ after executing a.

The *plain encoding* of $\mathcal{N}$ is shown in Figure 3.3 (b). It is obtained by replacing every read arc by a pair of directed flow arcs. The interleaving semantics of this net is identical to that of $\mathcal{N}$. However, its step semantics is not, because the aforementioned step $\{b, c\}$ cannot occur; it has to be divided into two steps $\{b\}$ and $\{c\}$ that can happen in either order.

The *place-replication encoding*, or PR-encoding [VSY98] is shown in Figure 3.3 (c) and remedies this situation. The structure of $\mathcal{N}$ is modified so that every place p in the context of $n \geq 1$ transitions $t_1, \ldots, t_n$ is substituted for places $p_1, \ldots, p_n$. Transitions producing or consuming p will produce or consume $p_1, \ldots, p_n$ instead. Moreover, transition t_i (for $i = 1, \ldots, n$) con-

sumes and produces place p_i instead of p. Thus, every reader of p has its own private copy of it. This preserves both the interleavings and the step semantics of $\mathcal{N}$ (in the sense of isomorphism).

This discussion shows that c-nets are no more expressive than Petri nets w.r.t. the usual semantics and could be seen as syntactic sugar for the PR-encoding. Notwithstanding this, treating c-nets directly has advantages for the unfolding approach, as we will see shortly.

3.1.3 Occurrence nets, unfoldings, and prefixes

We introduce a class of nets that satisfies certain acyclicity constraints, called occurrence nets. Fix a net $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$ for the rest of the section.

Definition 3.2 *The* causality relation *on $\mathcal{N}$, denoted $<$, is the least transitive relation over $P \cup T$ that includes F and satisfies $t < t'$ if $t^\bullet \cap \underline{t'} \neq \emptyset$, for all $t, t' \in T$. For $x \in P \cup T$, we write $[x]$ for the set of* causes *of x, defined as $\{ e \in E \mid e \leq x \}$, where $\leq$ is the reflexive closure of $<$.*

In Figure 3.3 (a) we have, e.g., $a < p$, $p < d$, and $a < b$. Occurrence nets are acyclic w.r.t. the relation $<$.

Definition 3.3 *$\mathcal{N}$ is called* occurrence net *if it satisfies the properties:*

(i) every place p has at most one producer, i.e. $|{}^\bullet p| \leq 1$;
(ii) the causal relation $<$ is irreflexive (hence $\leq$ is a partial order);
(iii) $<$-minimal places are the initial marking, i.e. $m_0 = \{ p \mid {}^\bullet p = \emptyset \}$;
(iv) for every transition t there is a reachable marking m that enables it.

Example 3.5 *Figure 3.2 and Figure 3.3 (a) show two occurrence nets. Figure 3.3 (b) and (c) are not occurrence nets, because their causality relation is not acyclic, and neither is Figure 3.4 because t_3 can never fire.*

Note that condition (iv) of Definition 3.3 deviates from usual definitions of occurrence nets (e.g., [EH08, BCM98, BBC+12]) in that it is an operational rather than a structural condition. We will clarify this structural definition in Section 3.2 and Section 3.3 for c-nets and Petri nets separately. In all cases, the idea is that for every transition there should be some execution that contains it. Also, since $<$ is acyclic no transition can fire twice in any execution.

We can now define the unfolding of $\mathcal{N}$. Intuitively, it is a safe, acyclic, and in general infinite c-net, where loops of $\mathcal{N}$ are "unrolled". Like before, this definition has an operational flavour:

Definition 3.4 *The* unfolding *of $\mathcal{N}$ is a tuple $\langle \mathcal{U}_{\mathcal{N}}, h \rangle$ consisting of an occurrence net $\mathcal{U}_{\mathcal{N}} := \langle B, E, G, D, \widehat{m}_0 \rangle$, and a mapping $h \colon (B \cup E) \to (P \cup T)$. (For convenience, we often equate an unfolding with its underlying net $\mathcal{U}_{\mathcal{N}}$.) We call the elements of B* conditions*, and those of E* events*; h maps conditions to places and events to transitions. We extend h to sets, multisets, and sequences in the usual way; h applied to a marking of $\mathcal{U}_{\mathcal{N}}$ (a set) yields a marking of $\mathcal{N}$ (a multiset).*

Conditions are tuples $\langle p, e' \rangle$, where $p \in P$ and $e' \in E \cup \{\bot\}$, and events are tuples $\langle t, M \rangle$, where $t \in T$ and $m \subseteq B$. We set $h(\langle p, e' \rangle) := p$ and $h(\langle t, m \rangle) := t$, respectively. A set m of conditions is called concurrent*, written $conc(m)$, if $\mathcal{U}_{\mathcal{N}}$ has a reachable marking m' s.t. $m' \supseteq m$.*

Then $\mathcal{U}_{\mathcal{N}}$ is the smallest net containing the following elements:

- *if $p \in m_0$, then $\langle p, \bot \rangle \in B$ and $\langle p, \bot \rangle \in \widehat{m}_0$;*
- *for any $t \in T$ and disjoint sets $m_1, m_2 \subseteq B$ with $conc(m_1 \cup m_2)$, $h(m_1) = {}^\bullet t$, and $h(m_2) = \underline{t}$, we have $e := \langle t, m_1 \cup m_2 \rangle \in E$, and for all $p \in t^\bullet$, we have $\langle p, e \rangle \in B$. Moreover, G and D are such that ${}^\bullet e = m_1$, $\underline{e} = m_2$, and $e^\bullet = \{ \langle p, e \rangle \mid p \in t^\bullet \}$.*

Example 3.6 *Figure 3.5 shows the unfoldings of the nets from Figure 3.3, where h is implicitly indicated by the labels of conditions and events.*

The net $\mathcal{U}_{\mathcal{N}}$ represents all possible behaviours of N, and in particular a marking m is reachable in N iff some $\widehat{m}$ with $h(\widehat{m}) = m$ is reachable in $\mathcal{U}_{\mathcal{N}}$. For the case where N is bounded, we are interested in computing an initial part of $\mathcal{U}_{\mathcal{N}}$ that has the same property.

Definition 3.5 *A set $X \subseteq E$ is called* causally closed *if $[e] \subseteq X$ for all $e \in X$. A* prefix *of $\mathcal{U}_{\mathcal{N}}$ is a net $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0 \rangle$ such that $E' \subseteq E$ is causally closed, $B' = \widehat{m}_0 \cup (E')^\bullet$, and G', D' are the restrictions of G, D to $(B' \cup E')$. The prefix $\mathcal{P}$ is called* marking-complete *if for all markings m of $\mathcal{N}$, m is reachable in $\mathcal{N}$ iff there exists a marking $\widehat{m}$ reachable in $\mathcal{P}$ such that $h(\widehat{m}) = m$.*

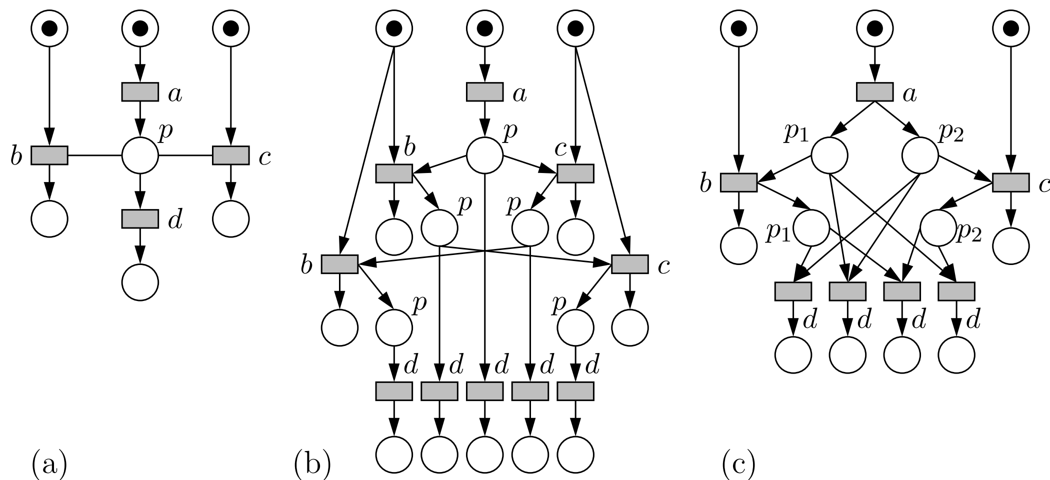

Figure 3.5: Unfoldings of the three nets from Figure 3.3.

In other words, a prefix is a causally closed subnet of $\mathcal{U}_{\mathcal{N}}$. It is known that if $\mathcal{N}$ is finite and bounded, then there exists a *finite* marking-complete prefix. A finite, marking-complete prefix then preserves all reachable markings of $\mathcal{N}$, while often being rather smaller than its reachability graph. For instance, in Figure 3.3 (a), if we replaced b, c by n transitions reading from p, then the net would have $\mathcal{O}(2^n)$ reachable markings, but the unfolding would still be isomorphic to the net itself. In general, the size of a marking-complete unfolding prefix is – asymptotically – somewhere between the size of the net and its reachability graph.

3.2 Making unfoldings more efficient

Unfoldings deal well with state-space explosion due to concurrent interleavings of independent actions because they simply leave independent actions as they are, without deciding their order of execution. This can be best seen in the example evoked in the beginning of the chapter, where we have n processes, each with one action that is entirely independent of the other processes. In this case, the interleaving semantics gives an LTS with 2^n states but the unfolding of the corresponding Petri net would be of size $\mathcal{O}(n)$.

Especially for the case of Petri nets, this has sparked a large body of research. Constructions of finite prefixes for safe or bounded Petri nets are

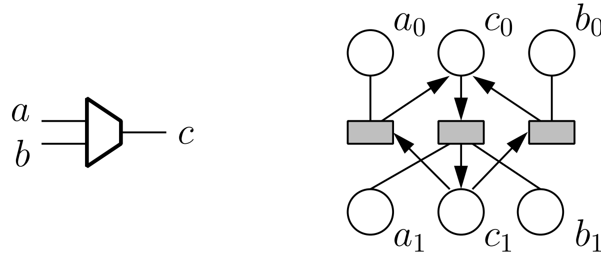

Figure 3.6: Logical AND-gate as a c-net.

found in [McM92, ERV02, KKV03], unfoldings-based verification algorithms [MR97, Hel99, KK00, EH01, ES01] and tools have emerged [Schb, Kho].

However, there are other sources of state-space explosion that traditional unfoldings do not cope well with. One is due to the fact that *concurrent read accesses* cannot be appropriately modeled in Petri nets and must be encoded indirectly, as we discussed in Section 3.1.2.

Consider again Figure 3.3, and let us call events labelled by b and c "readers", and events labelled by d "consumers". Intuitively, the readers are as independent as the n processes in our previous example. This is not recognized by traditional Petri net unfoldings, however, and the discussed encodings end up exploring every combination of different readers. Figure 3.5 shows the unfoldings of the nets from Figure 3.3. If we replaced b, c by n readers, there would be (a) n readers and one consumer in the contextual unfolding; (b) $\mathcal{O}(n!)$ readers *and* consumers in the plain unfolding; and (c) n readers but 2^n consumers in the PR-unfolding. Even for a minimal marking-complete prefix, the numbers would remain virtually the same, only in (b) the number of readers and consumers becomes $\mathcal{O}(2^n)$ instead of $\mathcal{O}(n!)$.

Read arcs are a natural extension of Petri nets. They have been used, e.g., to model concurrent database access [Ris94], concurrent constraint programs [MR94], priorities [JK91], and asynchronous circuits [VSY98]. Figure 3.6 shows the encoding of a logical AND-gate as a c-net.

Due to this, it seems reasonable to study a direct unfolding technique for c-nets. Unfoldings of c-nets were introduced independently by Baldan et al in [BCM98, BCM01] and Vogler et al in [VSY98]. The latter provided a first unfolding procedure for a restricted subclass called *read-persistent*. In this subclass, a net must not have a reachable marking enabling two transitions t, t' such that one reads from a place p and the other consumes it. This is a

severe restriction; for instance, the net in Figure 3.3 (a) does not satisfy it. A general but non-constructive procedure was proposed by Winkowski [Win02].

This section reviews the development of a general solution for c-net unfoldings, from the development of an abstract algorithm (Section 3.2.1) to concrete algorithms and data structures (Section 3.2.2), verification techniques (Section 3.2.3), and the combination with *merged processes* (Section 3.2.4). It represents joint work with Paolo Baldan, Alessandro Bruni, Andrea Corradini, Victor Khomenko, Barbara König, and César Rodríguez. While this author's contribution to the abstract algorithm was of secondary importance, his focus was on the development of the other components. The tool CUNF, which implements the techniques described herein, was made by César Rodríguez.

3.2.1 An abstract algorithm

In this section, which is based on [BCKS07, BCKS08], we develop a generic algorithm for constructing finite marking-complete prefixes for bounded c-nets. For the rest of the section, we fix a finite bounded c-net $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$ and its unfolding $\langle \mathcal{U}_\mathcal{N}, h \rangle$ with $\mathcal{U}_\mathcal{N} = \langle B, E, G, D, \widehat{m}_0 \rangle$.

We have seen that c-net unfoldings can be more compact than corresponding Petri net unfoldings. This advantage comes at the expense of a richer structure that contains some novel effects w.r.t. Petri net unfoldings.

Consider Figure 3.5 (a). Event d can happen after b has fired, or after c has fired, or both, or none. So event d "summarizes" these four situations that would be represented by multiple events in Figure 3.5 (b) and (c), and we say that d has multiple *histories*. These histories will play a central role in our algorithm. We give some definitions to capture them formally.

Definition 3.6 *[BCM98] Let $e, e' \in E$ be two events. We say e is a* direct causal predecessor *of e', written $e \lessdot e'$, if $e^\bullet \cap ({}^\bullet e' \cup \underline{e'}) \neq \emptyset$. We write $e \nearrow\!\!\!\nearrow e'$ (*direct asymmetric conflict*) if $\underline{e} \cap {}^\bullet e' \neq \emptyset$. Moreover, if $e \neq e'$ and ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$, then we say that e and e' are in* direct symmetric conflict*, written $e \#_i e'$.*

Finally, e, e' are in asymmetric conflict*, written $e \nearrow e'$, iff either (i) $e \lessdot e'$, or (ii) $e \nearrow\!\!\!\nearrow e'$, or (iii) $e \#_i e'$. For a set of events $X \subseteq E$, $\nearrow_X$ denotes the relation $\nearrow \cap (X \times X)$.*

In Figure 3.5 (a), we have $a \lessdot b$ and $a \lessdot d$ as well as $b \nearrow\!\!\!\nearrow d$. Direct symmetric conflict does not occur in (a) but in part (b) of the figure, e.g.,

between the topmost pair of events labelled b and c. All of these are also subsumed by the notion of asymmetric conflict. An asymmetric conflict implies a scheduling constraint: if both e, e' occur in a run, then e must occur first. In case (iii) this is vacuously true, as e, e' cannot both occur.

Definition 3.7 *A* configuration *of the unfolding $\mathcal{U}_\mathcal{N}$ is a finite, causally closed set of events $\mathcal{C}$ such that $\nearrow_\mathcal{C}$ is acyclic. $Conf(\mathcal{U}_\mathcal{N})$ denotes the set of all such configurations. The* cut *of a configuration $\mathcal{C}$ is the marking reached in $\mathcal{U}_\mathcal{N}$ by a run of $\mathcal{C}$, i.e. $\mathsf{Cut}(\mathcal{C}) := (\widehat{m}_0 \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$. The* marking *of $\mathcal{C}$ is its image through h: $\mathsf{Mark}(\mathcal{C}) := h(\mathsf{Cut}(\mathcal{C}))$.*

A set of events is a configuration iff all its events can be ordered to form a run that respects the scheduling constraints given by $\nearrow$. If such a configuration $\mathcal{C}$ contains a unique event e that must fire last, then we call $\mathcal{C}$ a *history* of e. For instance, in Figure 3.5 (a), the sets $\mathcal{C}_0 = \{a, d\}$, $\mathcal{C}_1 = \{a, b, d\}$, $\mathcal{C}_2 = \{a, c, d\}$, and $\mathcal{C}_3 = \{a, b, c, d\}$ are configurations. In all of them, d must fire last, so they are histories of d. In contrast, $\{a, b, c\}$ is a configuration but not a history for any event.

Definition 3.8 *Let $e \in E$ and $\mathcal{C}$ a configuration with $e \in \mathcal{C}$. We call the configuration $\mathcal{C}[\![e]\!] := \{\, e' \in \mathcal{C} \mid e'(\nearrow_\mathcal{C})^* e \,\}$ the* history *of e in $\mathcal{C}$. Moreover, $Hist(e) := \{\, \mathcal{C}[\![e]\!] \mid \mathcal{C} \in Conf(\mathcal{U}_\mathcal{N}) \wedge e \in \mathcal{C} \,\}$ is the set of histories of e.*

E.g., in Figure 3.5 (a), $Hist(d) = \{\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$. If $\mathcal{N}$ is a Petri net, then read arcs are absent and $Hist(e) = \{[e]\}$ is a singleton for all events of $\mathcal{U}_\mathcal{N}$.

The latter observation is important because it implies that the standard algorithm for obtaining a marking-complete prefix *for Petri nets* (e.g., [ERV02]) does not work. We briefly review this algorithm: it maintains the current prefix $\mathcal{P}$ and a set M of markings. Initially $\mathcal{P}$ contains only $\widehat{m}_0$ and $M = \{m_0\}$. Then it adds one event e at a time (applying the inductive part of Definition 3.4). If $\mathsf{Mark}([e]) \notin M$, then e and its postset are added to $\mathcal{P}$ and $\mathsf{Mark}([e])$ to M. Otherwise e is declared a *cutoff*, and neither e nor its causal successors are explored. It terminates when no more events can be added (due to cutoffs).

This procedure is implicitly parametrized by an order $\prec$ on configurations: If in one iteration, the algorithm has the choice between multiple events, it will pick one such that $[e]$ is minimal w.r.t. $\prec$. The precise details of $\prec$ are out of scope for this document. We merely remark that $\prec$ must be chosen

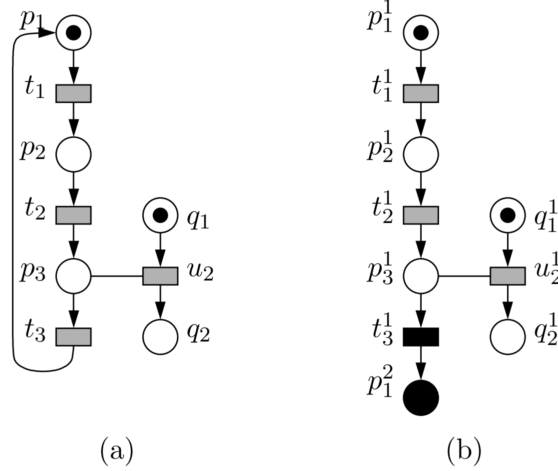

Figure 3.7: (a) A safe c-net; and (b) an incomplete unfolding prefix.

carefully to guarantee that one obtains a marking-complete prefix. Typically, one chooses a so-called *adequate* order [ERV02].

As Figure 3.7 shows, this procedure does not apply to c-nets. Part (a) of the figure shows a c-net, part (b) the prefix obtained with the procedure above (the choice of $\prec$ does not matter here), with the h-label of all conditions and events indicated in its name. Event t_3^1 is marked as cutoff because $\mathsf{Mark}([t_3^1])$ equals the initial marking. Hence, the marking $\{p_2, q_2\}$, reachable in (a), has no correspondence in (b). This shows that it is not sufficient to consider just $\mathsf{Mark}([e])$ for all events e. Instead, every event may contribute multiple reachable markings to the unfolding, one per history, and should be accepted as non-cutoff if at least one history contributes a new marking to the prefix. Notice, however, that events can have an infinite number of histories (because of read arcs), so this observation is not sufficient in itself.

The main contribution of [BCKS08] is an algorithm producing a marking-complete prefix of $\mathcal{U}_\mathcal{N}$ by lifting the procedure above to *enriched events*, i.e. pairs $\langle e, H\rangle$ such that $e \in E$ and $H \in \mathit{Hist}(e)$. A re-phrased version of that algorithm is given in Figure 3.8, where ξ represents the enriched events that are part of $\mathcal{P}$ and κ those that are cutoffs.

Algorithm 2, applied to Figure 3.7, would discover two histories for t_3^1, i.e. $H_1 = \{t_1^1, t_2^1, t_3^1\}$ and $H_2 = \{t_1^1, t_2^1, t_3^1, u_1^1\}$. Enriched event $\langle t_3^1, H_1\rangle$ will be declared cutoff (added to κ), whereas $\langle t_3^1, H_2\rangle$ is added to ξ. This gives rise to two more events t_1^2, t_2^2 (not shown) before the algorithm terminates.

Algorithm 2

Input: Bounded c-net $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$, ordering $\prec$
Output: $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0 \rangle$

Set $\widehat{m}_0 := \{ \langle p, \bot \rangle \mid p \in m_0 \}$; $B' := \widehat{m}_0$;
set $M := \{m_0\}$, $\xi := \emptyset$, and $\kappa := \emptyset$.
Repeat the following until termination:

1. Identify the set PE of enriched events $\langle e, H \rangle$ such that
   (i) $\langle e, H \rangle \notin \xi \cup \kappa$ and
   (ii) for all $e' \nearrow e$ with $e' \in H$ we have $\langle e', H[\![e']\!] \rangle \in \xi$.
2. If $PE = \emptyset$, terminate.
3. Choose and remove $\langle e, H \rangle$ from PE so that H $\prec$-minimal in PE.
4. if $\mathsf{Mark}(H) \in M$ then add $\langle e, H \rangle$ to κ and go to (i);
5. otherwise, add $\mathsf{Mark}(H)$ to M, add $\langle e, H \rangle$ to ξ;
   if e not yet in E, add e and $e^\bullet$ to $\mathcal{P}$ as per Definition 3.4.

Figure 3.8: Abstract algorithm for marking-complete c-net prefix

Proposition 3.1 summarizes the main results (mostly) of [BCKS08]:

Proposition 3.1 *If $\mathcal{N}$ is finite and bounded and $\prec$ is an adequate order in the sense of [ERV02], then Algorithm 2 terminates and produces a marking-complete prefix $\mathcal{P}$ of $\mathcal{U}_\mathcal{N}$.*

Notes on the proof: The statement follows from Theorems 1, 2, and 3 in [BCKS08]. Requirement 1.(ii) in Algorithm 2 is related to the notion of closedness (Definition 12, Lemma 1) and implies that any possible extension (member of PE) $\langle e, H \rangle$ can be constructed from other extended events already present in ξ. The order $\prec$ used in [BCKS08] is actually the partial order originally used by McMillan [McM92] where $\mathcal{C}_1 \prec \mathcal{C}_2$ iff $|\mathcal{C}_1| < |\mathcal{C}_2|$, which was shown to be inefficient for Petri nets in [ERV02]. The lifting to general adequate orders, allowing smaller prefixes, was done in [RSB11, BBC+12].

3.2.2 Efficient construction of c-net unfoldings

This section describes the endeavours made to develop an efficient, concrete algorithm for implementing Algorithm 2. It is based on publications [BBC+10, RSB11, BBC+12], where the latter is a journal article integrating the two former papers.

Algorithm 2 comes at a price: it is, a priori, rather more complex than comparable algorithms for Petri nets, e.g. [ERV02]. In particular, it requires to annotate every event e with some of its histories, i.e. sets of events. So it was not immediately clear whether the approach could be implemented with reasonable efficiency, and how. For the interesting subcase of *safe* c-nets, the interest of computing a complete contextual prefix was in question from a practical point of view: while the prefix can be exponentially smaller than the complete prefix of the corresponding PR-encoding, the intermediate structure used to produce it has asymptotically the same size (for bounded nets in general, this is not the case, see [BCKS08], Section 4). For instance, the c-net unfolding of Figure 3.5 (a) is much more compact than the unfolding of the PR-encoding in Figure 3.5 (c), but event d in the former has four histories corresponding to the four d-labelled events in the latter.

Let us fix a finite, bounded c-net $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$ and its unfolding $\langle \mathcal{U}_\mathcal{N}, h \rangle$, where $\mathcal{U}_\mathcal{N} = \langle B, E, G, D, \widehat{m}_0 \rangle$, as in Section 3.2.1.

The main challenges to building an efficient unfolder were the following:

(I) to identify the set PE of *possible extensions* $\langle e, H \rangle$ in step 1 of Algorithm 2;

(II) which data structures to use to store and process histories efficiently and compactly.

Problem (I) has two subproblems: how to identify an event e, and how to identify a history H. For the first, Definition 3.4 tells us to identify two sets of conditions m_1, m_2 such that $conc(m_1 \cup m_2)$ and suitable other properties hold. In the case of Petri nets, it is known that $conc(m)$ holds iff $conc(\{p, q\})$ holds for all pairs $p, q \in m$. Unfortunately this is not the case for c-nets, as Figure 3.9 shows: any two conditions from the set $m = \{d_1, d_2, d_3\}$ are reachable by firing two appropriate events among e_1, e_2, e_3, but $conc(m)$ does not hold.[1]

[1]Incidentally, $\{e_1, e_2, e_3\}$ *would* be a step in the semantics of [JK91] – see Section 3.1 – hence $conc(m)$ would hold under that alternative semantics. This is not a general phenomenon, however.

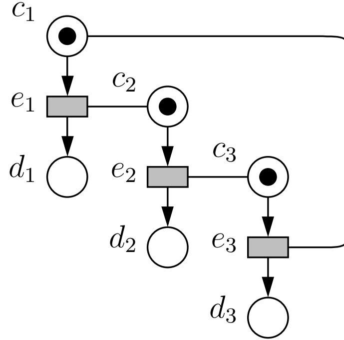

Figure 3.9: Example showing that $conc(\cdot)$ is not a binary relation for c-nets.

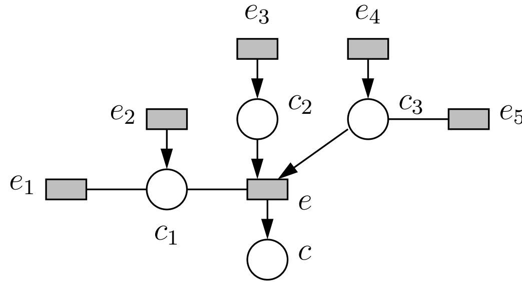

Figure 3.10: Asymmetric-conflict predecessors of e for history construction.

As for identifying a history of a given event e, step 1.(ii) of Algorithm 2 gives us a hint: it should decompose into (and hence can be constructed from) already identified extended events for the $\nearrow$-predecessors of e. Consider Figure 3.10 and its elements in relation to event e. Event e_1 does not satisfy $e_1 \nearrow^* e$ and therefore *must not* be part of any history of e. Events e_2, e_3, e_4 are causal precedessors ($<$) of e and therefore *must* be included in any history, more precisely one of their histories must be included in H. As for e_5, it is not a causal predecessor of e but $e_5 \not\nearrow e$ holds. Thus e_5 (and one of its histories) *can* be included in H.

This hints at a solution for problem (II): a history H for e can be stored in memory by pointing from e to the histories of its $\nearrow$-predecessors used to construct H. This data structure, which we call *history graph*, allows to share common parts of different histories and has other useful properties; we refer the reader to [BBC+12], Section 6.1 for details.

As for problem (I), the solution for both subproblems at once is the

concept of *enriched conditions*: a condition c together with the history of its producing event, its readers, or some combination thereof. Let us first classify relations between configurations:

Definition 3.9 *Let $\mathcal{C}, \mathcal{C}' \in Conf(\mathcal{U}_{\mathcal{N}})$. We write $\mathcal{C} \sqsubseteq \mathcal{C}'$ ($\mathcal{C}$ can* evolve *into $\mathcal{C}'$) if (i) $\mathcal{C} \subseteq \mathcal{C}'$ and (ii) no pair $e_1 \in \mathcal{C}, e_2 \in \mathcal{C}' \setminus \mathcal{C}$ satisfies $e_2 \nearrow e_1$. Moreover, $\mathcal{C}, \mathcal{C}'$ are said to be in* conflict*, written $\mathcal{C} \# \mathcal{C}'$, when there is no configuration $\mathcal{C}''$ verifying $\mathcal{C} \sqsubseteq \mathcal{C}''$ and $\mathcal{C}' \sqsubseteq \mathcal{C}''$.*

Interestingly, $\sqsubseteq$ is not simply subset inclusion: for instance, in Figure 3.5 (a), $\{a, d\}$ and $\{a, b, d\}$ are configurations, but $\{a, d\} \sqsubseteq \{a, b, d\}$ does not hold: once d is fired, it is too late for b. In this case, $\{a, d\} \# \{a, b, d\}$ holds; both configurations cannot converge to a common future.

Definition 3.10 *Let c be a condition. A* generating history *of c is $\emptyset$ if $c \in \widehat{m}_0$, or $H \in Hist(e)$, where $\{e\} = {}^{\bullet}c$. A* reading history *of c is any $H \in Hist(e)$ such that $e \in \underline{c}$. A* history *of c is any of its generating or reading histories or $H_1 \cup H_2$, where H_1 and H_2 are histories of c verifying $\neg(H_1 \# H_2)$. In the latter case, the history is called* compound*. For any history H of c, the pair $\langle c, H\rangle$ is called* enriched condition*.*

For instance, in Figure 3.5 (a), condition p has four histories: $\{a\}$ is generating, $\{a, b\}$ and $\{a, c\}$ are reading, and $\{a, b, c\}$ is compound.

We now introduce a binary relation between enriched conditions, which has useful properties as we will soon see.

Definition 3.11 *Two enriched conditions $\langle c, H\rangle, \langle c', H'\rangle$ are called* concurrent*, written $\langle c, H\rangle \parallel \langle c', H'\rangle$, iff $\neg(H \# H')$ and $c, c' \in \mathsf{Cut}(H \cup H')$.*

Intuitively, if $\langle c, H\rangle \parallel \langle c', H'\rangle$, then H, H' have a common future and neither of them consumes c or c'. E.g., in Figure 3.9, we have:

- $\langle c_1, \emptyset\rangle \not\parallel \langle d_1, \{e_1\}\rangle$ because $c_1 \notin \mathsf{Cut}(\{e_1\})$;
- $\langle d_1, \{e_1\}\rangle \not\parallel \langle d_2, \{e_2\}\rangle$ because $\{e_1\} \# \{e_2\}$;
- $\langle c_1, \emptyset\rangle \parallel \langle d_3, \{e_3\}\rangle$ and $\langle d_1, \{e_1\}\rangle \parallel \langle d_2, \{e_1, e_2\}\rangle$.

The relation $\parallel$ turns out to have useful properties to solve problem (I): let $m = \{c_1, \ldots, c_n\}$ be a set of conditions, then there exists a set of enriched conditions $X = \{\langle c_1, H_1\rangle, \ldots, \langle c_n, H_n\rangle\}$, with $\rho \parallel \rho'$ for any pair $\rho, \rho' \in X$, if and only if $conc(m)$. Moreover, the existence of X implies that $H_1 \cup \cdots \cup H_n$ is a configuration.

All the above amounts to saying that to obtain an enriched event $\langle e, H\rangle$, one identifies (i) one history H_c for each $c \in {}^\bullet e$ and (ii) one *generating* history H_c for each $c \in \underline{e}$, such that $\langle c, H_c\rangle \parallel \langle c', H_{c'}\rangle$ holds for all pairs $c, c' \in {}^\bullet e \cup \underline{e}$. The union of these histories plus e itself then is a history for e.

We developed two basic approaches for choosing such histories, each with one variation. Here, we only give a short summary and state the most elegant of the formulations, which is also the one used by default in the tool Cunf.

- In [BBC+10], we developed an approach we later called *lazy*. There, only generating and reading histories are kept in memory, and compound histories are constructed 'on demand', i.e. in case (i) above a compound history for $c \in {}^\bullet e$ is represented as a set of reading histories.

- In [RSB11], we developed the *eager* approach: it keeps all enriched histories in memory, including compound. Thus, one literally chooses one history represented as such for every $c \in {}^\bullet e$ in (i) above.

The journal version, [BBC+12], presents in Section 5 a synthesis of the two approaches, which represent a time/space tradeoff. Their relative merits are discussed in [BBC+12], Section 5.4, and we give constructed examples where one approach outperforms the other in both directions.

There is one further catch: the same enriched event $\langle e, H\rangle$ may be obtained through (many) different combinations of enriched conditions. Consider the unfolding in Figure 3.11. Condition c has $n + 1$ different reading histories: $H_0 := \emptyset$, $H_1 := \{e_1\}$, $\ldots$, $H_n := \{e_1, \ldots, e_n\}$, while c' has one single history $H := H_n$, and $\langle c, H_i\rangle \parallel \langle c', H\rangle$ holds for all $i = 0, \ldots, n$. However, in all cases, $H_i \cup H = H$, so there exists a multitude of possibilities to construct the enriched event $\langle e, H \cup \{e\}\rangle$.

Since such ambiguity causes needless exploration work in an unfolding tool, we propose amendments for both approaches to allow one single decomposition per event in [BBC+12], Section 5.3. For the lazy approach, this entails the introduction of another relation called *subsumption* ($\propto$), while the eager approach admits an elegant representation with just a single relation that we call *asymmetric concurrency*. Here, we summarize only the latter:

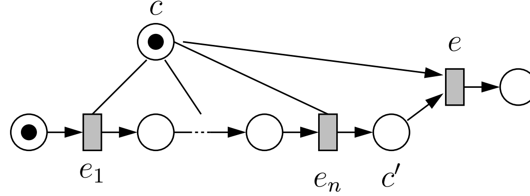

Figure 3.11: Example where the same history for e can be composed in multiple ways.

Definition 3.12 *Let $\rho = \langle c, H\rangle$ and $\rho' = \langle c', H'\rangle$ be two enriched conditions. We say that ρ is* asymmetrically concurrent *to ρ', written $\rho \mathbin{/\!/} \rho'$ iff $\rho \parallel \rho'$ and $\underline{c} \cap H' \subseteq H$.*

Notice that $/\!/$ is an asymmetric relation, saying that all readers of c in H' must be in H. In Figure 3.11, it implies, e.g., that $\langle c, H_n\rangle \mathbin{/\!/} \langle c', H_n\rangle$, but $\neg(\langle c, H_i\rangle \mathbin{/\!/} \langle c', H_n\rangle)$ for any $i < n$. As a result, we obtain a unique decomposition for each possible extension. The following proposition is stated as Corollary 2 in [BBC+12].

Proposition 3.2 *The pair $\langle e, H\rangle$ with $h(e) = t$ is an enriched event iff there exist sets X_p, X_c of enriched conditions such that*

1. *$h(X_p) = {}^\bullet t$ and $h(X_c) = \underline{t}$;*
2. *X_p contains arbitrary enriched conditions, X_c generating conditions;*
3. *$X_p \cup X_c$ contains exactly one enriched condition for every $c \in ({}^\bullet e \cup \underline{e})$;*
4. *$\rho \mathbin{/\!/} \rho'$ for $\rho \in X_p$ and $\rho' \in X_p \cup X_c$;*
5. *$\rho \mathbin{/\!/} \rho'$ or $\rho' \mathbin{/\!/} \rho$ for all $\rho, \rho' \in X_c$;*
6. *finally, $H = \{e\} \cup \bigcup\{\, H' \mid \langle c, H'\rangle \in X_p \cup X_c \,\}$.*

Moreover, for any enriched event $\langle e, H\rangle$ there exists exactly one pair of sets X_p, X_c satisfying properties 1–6.

Finally, an unfolding tool can efficiently identify possible extensions if the relations $\parallel$ and $/\!/$ (or $\propto$) are kept in memory. This calls for fast procedures to compute and incrementally update the relations whenever the current

prefix is extended by new (extended) events and conditions. This is indeed possible and usually amounts to a small number of set operations per event. While the existence of these operations is crucial for the efficient operation of an unfolding tool, we refrain from a detailed presentation at this point; the reader is referred to Propositions 4, 5, 7, and 8 in [BBC+12].

Finally, the techniques were implemented (by César Rodríguez) in an unfolding tool called CUNF, which operates on *safe* c-nets (the benchmarks we were interested in were all safe). Much attention was given to optimizing the implementation. In [RSB11] and [BBC+12], Section 7, we report on experiments. Some of the main conclusions are summarized below:

- As expected, the runtime performance of the lazy approach degraded in the presence of many compound conditions. The memory savings in those cases (ca. 60%) did not seem to outweigh the loss in run-time, which was in orders of magnitude. Therefore, we preferred the eager approach.

- The eager approach consistently outperformed the unfolding of the PR-encoding in terms of runtime.

- The eager approach and the unfolding of the plain encoding handle all examples from a standard set of benchmarks gracefully, but the resulting c-net unfoldings can represent the state space much more concisely.

Moreover, we conducted encouraging experiments with asynchronous circuits. Logic gates have a natural encoding in c-nets (cf. Figure 3.6). We give a simple example to illustrate their benefits. We consider a grid of $n := k \times k$ AND-gates, shown in Figure 3.12 for $k = 3$. The inputs for the AND-gates are at the left and top of the figure, and outputs propagate to the right and towards the bottom. Inputs may switch freely between high and low. We then used CUNF to construct complete unfolding prefixes of the corresponding c-nets and their plain encodings, and observed that signal changes may be propagated to the bottom right in many different orders, which are distinguished by Petri-net unfoldings but not by c-net unfoldings. Hence, unfoldings of the plain nets were of exponential size in n, while the contextual ones were linear. Moreover, CUNF built the latter ones in time $\mathcal{O}(n^3)$, see Figure 3.12; in this case, also the PR-encoding is efficient.

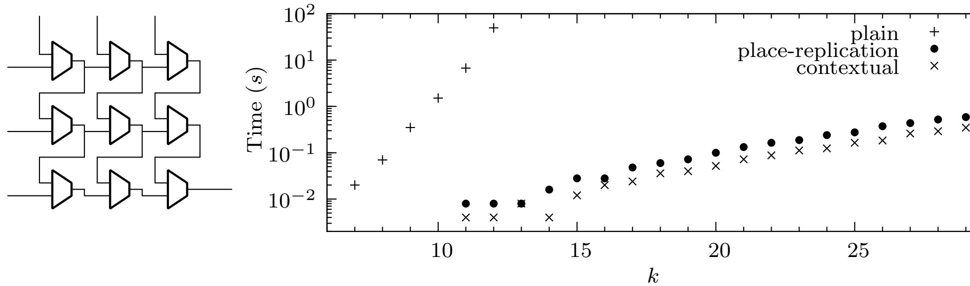

Figure 3.12: Unfolding times for the plain, PR, and contextual net encodings of the AND-gate networks of size $n := k \times k$.

3.2.3 Verification of c-net unfoldings

Having achieved an efficient construction to obtain a finite, marking-complete prefix $\mathcal{P}$ for a c-net $\mathcal{N}$, we now discuss how $\mathcal{P}$ can help to verify properties of $\mathcal{N}$. The material in this section is a brief summary of [RS12].

Let $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$ be a net, $\langle \mathcal{U}_\mathcal{N}, h \rangle$ its unfolding, and $\mathcal{P} = \langle B, E, G, D, \widehat{m}_0 \rangle$ a prefix of $\mathcal{U}_\mathcal{N}$. For simplicity we assume that $\mathcal{N}$ is *safe*.[2]

As stated before, the initial interest in unfoldings lay in the fact that, while $\mathcal{P}$ is in general larger than $\mathcal{N}$, it is in general smaller than the reachability graph. Moreover, it is known [McM92] that deadlock or reachability checking are PSPACE-complete for $\mathcal{N}$ but NP-complete for $\mathcal{P}$.[3] Thus, the unfolding technique represents a time/space tradeoff for verifying c-nets. This tradeoff is particularly attractive when testing multiple properties of the same net because $\mathcal{P}$ needs to be constructed only once.

In the past, various efforts have been made to exploit this fact for the case of Petri nets, using reductions to different NP-complete problems: McMillan [McM92] employed a branch-and-bound technique, Heljanko [Hel99] a stable-models encoding, and Melzer and Römer [MR97] used mixed integer linear programming, later improved by Khomenko and Koutny [KK00, KK07]. Esparza and Schröter [ES01] devised an ad-hoc algorithm based on additional information obtained while computing the unfolding.

Given the emergence of powerful SAT solvers like Minisat [ES03] since

[2]Section 4.5 in [RS12] outlines how to extend to the case of bounded nets.

[3]McMillan's result holds for the case of Petri nets (without read arcs), but the hardness result trivially holds for c-nets, too. For the upper bound, this section provides polynomial encodings from $\mathcal{P}$ into SAT.

then, we prefer to reduce to SAT instead. The encoding is also natural because unfoldings are safe nets, so the marking of a place naturally translates to a boolean variable. Indeed, SAT solving has already been proposed for the similar problem of model-checking merged processes [KKKV06], and [EH08] gives an explicit SAT encoding for Petri net unfoldings. In [RS12], we lifted these encodings to c-nets. This is briefly sketched below with some optimizations. As for the latter, we concentrate on optimizations that exploit structural information of the net, while leaving optimizations at the logical level to the SAT solvers (which are extremely good at them).

The SAT problem is as follows: given a formula ϕ of propositional logic in conjunctive normal form (CNF), find whether there exists a satisfying assignment that makes ϕ true. We discuss the following problem:

> **Deadlock checking:** Does $\mathcal{N}$ have a deadlock, i.e. a marking in which no transition can fire? This amounts to asking whether $\mathcal{P}$ contains a configuration $\mathcal{C}$ with no transition enabled in $\mathsf{Mark}(\mathcal{C})$. We construct a formula $\phi_{\mathcal{P}}^{\text{dead}}$ that is satisfiable iff such a configuration exists.

The formulae for properties like reachability or coverability are very similar, and we refer the reader to [RS12], Section 4.4, or [BBC+12], Section 8.

$\phi_{\mathcal{P}}^{\text{dead}}$ is defined over variables e for $e \in E$ and p for $p \in P$ as:

$$\phi_{\mathcal{P}}^{\text{dead}} := \phi_{\mathcal{P}}^{\text{causal}} \wedge \phi_{\mathcal{P}}^{\text{sym}} \wedge \phi_{\mathcal{P}}^{\text{asym}} \wedge \phi_{\mathcal{P}}^{\text{mark}} \wedge \phi_{\mathcal{P}}^{\text{dis}}$$

The first three constraints enforce a satisfying assignment to represent a configuration $\mathcal{C}$, and $\phi_{\mathcal{P}}^{\text{mark}}$ extracts $\mathsf{Mark}(\mathcal{C})$, which $\phi_{\mathcal{P}}^{\text{dis}}$ verifies to be deadlocked. Most of these subformulae are straightforward and we omit them here, except the most interesting one, $\phi_{\mathcal{P}}^{\text{asym}}$.

Asymmetric conflict loops $\phi_{\mathcal{P}}^{\text{asym}}$ treats conditions (i) and (ii) of Definition 3.6, i.e. $e \lessdot e'$ and $e \nearrow e'$. Let us define the relation $R := \lessdot \cup \nearrow$. Our task is to identify cycles in the relation R. Note that this requirement is new w.r.t. encodings for Petri nets, which cannot contain such cycles.

Example 3.7 *We first demonstrate that such cycles occur naturally in well-known examples: Consider Figure 3.13, which shows the beginning of an unfolding of Dekker's mutual-exclusion algorithm [Ray86] (only some events of interest are shown). In the beginning, both processes indicate their interest to enter the critical section by raising their flag (events e_1, f_1). They then*

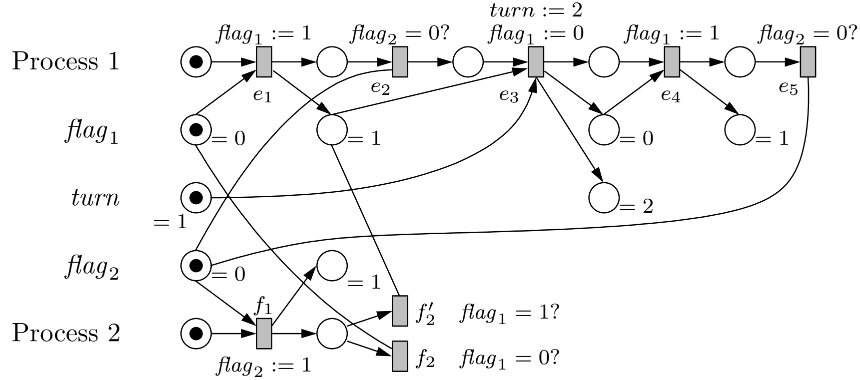

Figure 3.13: Partial unfolding of Dekker's algorithm algorithm with asymmetric cycles.

check whether the flag of the other process is low (events e_2, f_2*) and if so, proceed (*e_3*) and possibly repeat (*e_4, e_5*). If both processes want to enter the critical section (*h_2'*), some arbitration happens (not displayed). Two conflict cycles in this example are* $e_1 \lessdot e_2 \nearrow\!\!\!\nearrow f_1 \lessdot f_2 \nearrow\!\!\!\nearrow e_1$ *and* $f_1 \lessdot f_2' \nearrow\!\!\!\nearrow e_3 \lessdot e_4 \lessdot e_5 \nearrow\!\!\!\nearrow f_1$*.*

Several encodings have been proposed in the literature for acyclicity constraints [CGS09] that are of size $\mathcal{O}(n^2)$ or $\mathcal{O}(n \cdot \log n)$, for $n := |R|$. Since this is by far the most costly part of the encoding, we discuss some optimizations in [RS12], Section 4.1, the most interesting of which are:

- Any cycle in R must contain at least two instances of $\nearrow\!\!\!\nearrow$. So we can replace R by the relation $R' := \{\, (e, g) \mid \exists f, h : e \nearrow\!\!\!\nearrow f \leq g \nearrow\!\!\!\nearrow h \,\}$, which contains a cycle iff R does.

- We operate in two phases: first, we invoke the SAT checker *without* the constraint $\phi_{\mathcal{P}}^{\text{asym}}$. This may result in false positives, i.e. a configuration containing a cycle in R. If the SAT solver comes up with such a spurious deadlock, we repeat with $\phi_{\mathcal{P}}^{\text{asym}}$ properly included. This simple trick was very useful, with only 2% false positives in over 100 examples.

Reduction of stubborn events We discuss one other optimization that palliates a problem of SAT checkers. Consider the occurrence net shown in Figure 3.14. If event e_1 fires, then nothing can prevent e_2, e_3, e_4, and e_5 from

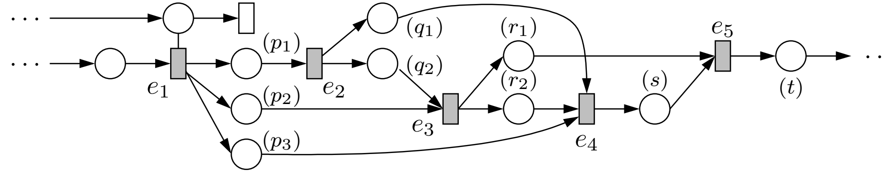

Figure 3.14: Stubborn events.

firing. Thus, any configuration leading to a deadlock must either contain all five events or none of them. However, e_1 is not guaranteed to fire due to the white event that consumes from its context.

Modern SAT solvers are very efficient at *unit propagation*, i.e. known or tentative truth values are propagated to simplify other clauses [ES03]. In the SAT encoding for Figure 3.14, a SAT solver can immediately decide that e_2 must fire when e_1 does. However, unit propagation is not able to detect that e_3 and e_4 are logical implications of e_1. When an unfolding contains many such situation, the SAT checker becomes unnecessarily inefficient.

On the other hand, such information is easy to detect on the unfolding structure. Before calling the SAT solver, we identify *stubborn* events e, i.e. those satisfying $({}^\bullet e \cup \underline{e})^\bullet = \{e\}$. Intuitively, once all events preceding e have fired, then firing e is unavoidable to find a deadlock, and in Figure 3.14, events e_2, e_3, e_4, e_5 are all stubborn. We eliminate such stubborn events from $\phi_{\mathcal{P}}^{\text{dead}}$ by 'merging' them appropriately with their predecessors. The details are given in [RS12], Section 4.2.

Experimental evaluation Section 5 of [RS12] reports on experimental evaluation. Of course, the advantage of c-nets over Petri nets becomes arbitrarily large when using examples where the compactness advantages of the former are fully realized (e.g., in cases like Figure 3.5 or Figure 3.12). We also study a set of standard examples commonly used in the unfolding literature and compare against the best known previous method for unfoldings-based deadlock checking [KK07]. Even for these examples, which work already very well for traditional unfolding techniques, we obtain a time saving of 30% overall (unfolding+verification), and a speed-up factor of 2 for the verification phase alone.

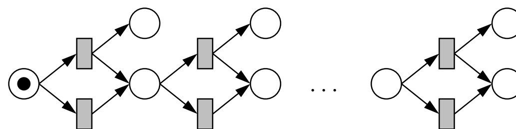

Figure 3.15: A Petri net with exponentially large unfolding prefix.

3.2.4 Contextual merged processes

This section describes the integration of c-net unfoldings with *merged processes*. It reports on work published in [RSK13].

As we have seen, contextual net unfoldings cope well with two sources of state-space explosion: concurrency and concurrent read accesses. Recently, a data structure has emerged that deals with a third source of explosion, i.e. *choices*. Consider Figure 3.15 and suppose that there are n pairs of transitions. There is a conflict between each pair of transitions, and they lead to different markings. Thus, there will be no cutoffs, and the smallest marking-complete unfolding prefix for that net will be of size $\mathcal{O}(2^n)$.

In [KKKV06], Khomenko et al provided a remedy for this problem, called *merged processes*. The idea behind these is to fuse certain conditions and events of a Petri net unfolding together, according to criteria that we will detail in a moment. For instance, the merged process of the net in Figure 3.15 is isomorphic to that net. From a practical point of view, this makes perfect sense – the resulting structure is acyclic, like an unfolding, and therefore enjoys some of the same advantages as the unfolding itself; e.g., reachability can be polynomially encoded as a SAT formula. In [KM11], Khomenko and Mokhov presented a direct construction method for merged processes.

Although merged processes and c-nets are two complementary techniques that attack different sources of state-space explosion, there are some striking similarities: a single event may have multiple different "histories", and as we shall see, merging may create cycles in the flow relation, similar to the cycles in the asymmetric conflict relation of c-nets; yet, in a marking-complete merged process every marking remains reachable through a non-cyclic firing sequence. Hence, the combination of the two techniques is not only possible, but also is very natural, and in the remainder of the section, we will present such a combination, called *contextual merged processes*, or CMPs. A central notion is that of occurrence depth:

Definition 3.13 *Let x be a condition or event of some occurrence net. The*

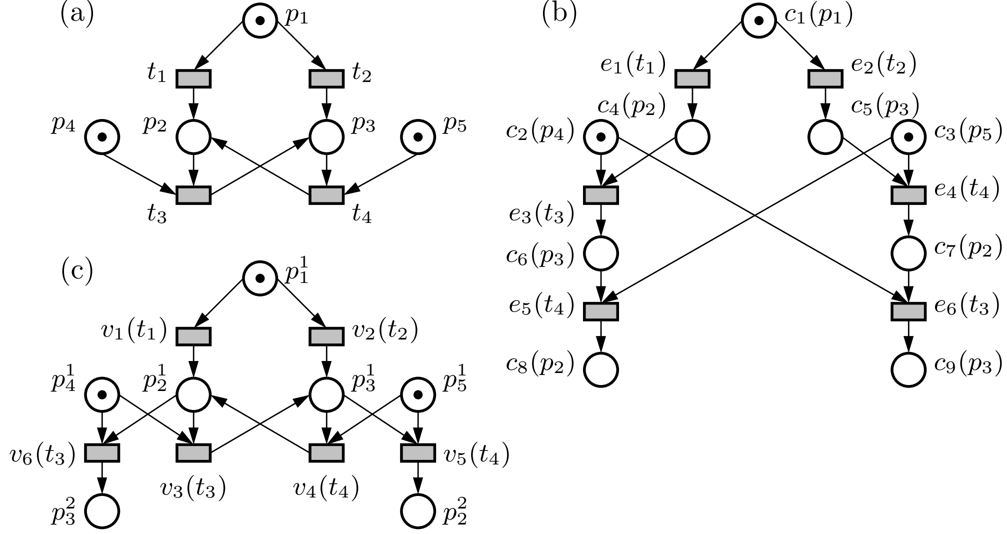

Figure 3.16: (a) A net; (b) its unfolding; (c) its unravelling.

occurrence depth *of* x*, denoted* $\mathrm{od}(x)$*, is the maximum number of* $h(x)$*-labelled nodes in any path in the directed graph* $\langle \tilde{m}_0 \cup [x] \cup [x]^\bullet, \lessdot \rangle$ *starting at any initial condition and ending in* x*.*

This relation is identical to the one given in [KKKV06, KM11] except that the relation $\lessdot$ also takes into account causal dependencies from read arcs. As an example, consider Figure 3.16 (b), which contains the unfolding of the net in Figure 3.16 (a). Among the p_2-labelled conditions, $\mathrm{od}(c_4) = \mathrm{od}(c_7) = 1$ and $\mathrm{od}(c_8) = 2$.

A CMP is now obtained in two steps: first, one fuses together all conditions that have the same label and occurrence depth, and secondly, one then fuses together all events that have the same environment (preset, context, postset) after the first step.

Definition 3.14 *Let* $\mathcal{N} = \langle P, T, F, C, m_0 \rangle$ *be a c-net and denote its unfolding or a prefix of it as* $\mathcal{P} = \langle \langle B, E, G, D, \tilde{m}_0 \rangle, h \rangle$*. Define a net* $\mathcal{Q} = \langle \widehat{B}, \widehat{E}, \widehat{G}, \widehat{D}, \widehat{m}_0 \rangle$*, where* $\widehat{B} \subseteq P \times \mathcal{N}$*,* $\widehat{E} \subseteq T \times 2^{\widehat{B}} \times 2^{\widehat{B}} \times 2^{\widehat{B}}$*, and a mapping* $\hbar : B \cup E \to \widehat{B} \cup \widehat{E}$ *as follows:*

- *for* $b \in B$*,* $\hbar(b) := \langle h(b), \mathrm{od}(b) \rangle$*; set* $\widehat{B} := \hbar(B)$*;*
- *for* $e \in E$*,* $\hbar(e) := \langle h(e), \hbar({}^\bullet e), \hbar(\underline{e}), \hbar(e^\bullet) \rangle$*; set* $\widehat{E} := \hbar(E)$*;*

- $\widehat{G}, \widehat{D}$ *are such that for every* $\widehat{e} = \langle t, Pre, Cont, Post\rangle \in \widehat{E}$ *we have* ${}^\bullet\widehat{e} := Pre$, $\underline{\widehat{e}} := Cont$, $\widehat{e}^\bullet := Post$;

- $\widehat{m}_0(\langle p, d\rangle) := |\tilde{m}_0 \cap \{\, b \in B \colon h(b) = p,\ \mathrm{od}(b) = d \,\}|$.

Moreover, let $\widehat{h} : \widehat{B} \cup \widehat{E} \to P \cup T$ *be the mapping of the node from* $\mathcal{Q}$ *to* $\mathcal{N}$ *given by projecting the nodes of* $\mathcal{Q}$ *to their first components. We call* $\mathfrak{Merge}(\mathcal{P}) := \langle \mathcal{Q}, \widehat{h}\rangle$ *the* merged process *of* $\mathcal{P}$*. The merged process* $\mathcal{M}_{\mathcal{N}} := \mathfrak{Merge}(\mathcal{U}_{\mathcal{N}})$ *of the unfolding of* $\mathcal{N}$ *is called the* unravelling *of* $\mathcal{N}$*.*

Example 3.8 *[KKKV06] Figure 3.16 (c) shows the merged process of the unfolding in Figure 3.16 (b). Here, we fuse the pairs* c_4, c_7 *and* c_5, c_6 *in the first step, where conditions* $\langle p, d\rangle$ *are written* p^d*. No events are fused in the second step in this example. Instead, there are two copies each of* t_3 *and* t_4 *that differ only in the occurrence depth of the conditions that they output to.*

Example 3.8 demonstrates a consequence of merging: in general, a merged process is not acyclic. For instance, the unravelling $\mathcal{M}_{\mathcal{N}}$ in Figure 3.16 (c) admits the firing sequence $v_1v_3v_4$, ending up with a token on p_2^1. We remark that the corresponding events in the unfolding $\mathcal{U}_{\mathcal{N}}$, $e_1e_3e_4$, are not a valid firing sequence. On the other hand, the non-cyclic sequence $v_1v_3v_5$ rectifies the situation: one ends up with a token on p_2^2, and $e_1e_3e_5$ is a valid firing sequence in $\mathcal{U}_{\mathcal{N}}$.

Moreover, not even all *acyclic* firing sequences in $\mathcal{M}_{\mathcal{N}}$ correspond to valid sequences in $\mathcal{U}_{\mathcal{N}}$. For instance, the firing sequence v_1v_6 is possible in $\mathcal{M}_{\mathcal{N}}$ but e_1e_6 is not possible in $\mathcal{U}_{\mathcal{N}}$. Indeed, the sequence v_1v_6 does not "feel right" because it puts a token onto p_3^2, i.e. the copy of p_3 with occurrence depth 2, before having put a token onto p_3^1.

These examples pose the question whether we can give a characterization of the "reasonable" executions in $\mathcal{M}_{\mathcal{N}}$. Definition 3.15 shall answer this question. Fix $\mathcal{N} = \langle P, T, F, C, m_0\rangle$ as a bounded c-net, $\langle \mathcal{U}_{\mathcal{N}}, h\rangle$ as its unfolding, with $\mathcal{U}_{\mathcal{N}} = \langle B, E, G, D, \widehat{m}_0\rangle$, and $\langle \mathcal{Q}_{\mathcal{N}}, \widehat{h}\rangle$, where $\mathcal{Q}_{\mathcal{N}} = \langle \widehat{E}, \widehat{B}, \widehat{G}, \widehat{D}, \widehat{m}_0\rangle$ is the corresponding merged process, i.e. $\mathfrak{Merge}(\mathcal{P}_N)$. The places of $\mathcal{Q}_N$ are called *mp-conditions* and its transitions *mp-events*.

Definition 3.15 *A multiset of mp-events* $\widehat{\mathcal{C}}$ *is an* mp-configuration *of* $\mathcal{Q}_{\mathcal{N}}$ *if there exists a configuration* $\mathcal{C}$ *of* $\mathcal{U}_N$ *verifying* $\hbar(\mathcal{C}) = \widehat{\mathcal{C}}$*.*

Example 3.9 *According to Definition 3.15,* $\{v_1, v_3, v_5\}$ *is an mp-configuration but not, e.g.,* $\{v_1, v_6\}$ *or* $\{v_1, v_3, v_4\}$*.*

The main technical contribution in [RSK13] is Proposition 3.3 that characterize mp-configurations in structural terms rather than indirectly via $\mathcal{U}_{\mathcal{N}}$, for the case of *safe* nets. (This problem is still open for bounded nets even in the case without read arcs.) Such a characterization is important in practice – it is a necessary building block to construct merged processes directly and also required, e.g., for verification purposes – recall in Section 3.2.3 we relied on encoding a configuration in SAT.

Proposition 3.3 *If N is safe, a set of mp-events $\widehat{\mathcal{C}}$ is an mp-configuration of $\mathcal{Q}_{\mathcal{N}}$ iff it satisfies the following conditions:*

1. $\forall \widehat{e} \in \widehat{\mathcal{C}} : \forall \widehat{c} \in {}^\bullet\widehat{e} \cup \underline{\widehat{e}} : (\widehat{c} \in \widehat{m}_0 \vee \exists \widehat{e}' \in {}^\bullet\widehat{c} : \widehat{e}' \in \widehat{\mathcal{C}})$*, and*
2. $\nearrow_{\widehat{\mathcal{C}}}$ *is acyclic, and*
3. *for $k \geq 1$, $p^{k+1} \in \widehat{\mathcal{C}}^\bullet$ implies $p^k \in \widehat{m}_0 \cup \widehat{\mathcal{C}}^\bullet$ and there exists a path in the directed graph $(\widehat{m}_0 \cup \widehat{\mathcal{C}} \cup \widehat{\mathcal{C}}^\bullet, <_i)$ between p^k and p^{k+1}.*

The first constraint in Proposition 3.3 imposes an adaptation of causal closure. Unsurprisingly, given the previous examples, the other two constraints forbid cycles and impose a "no gap" constraint, e.g. it is not allowed for a run to put a token on p^{k+1} without having visited p_k before, for some mp-condition p and $k > 0$:

A key detail in both results is that acyclicity of $\nearrow$ prohibits, at the same time, asymmetric conflicts inherent to c-net unfoldings (Figure 3.9) and cycles in the flow relation introduced by merging (Figure 3.16 (c)).

We currently do not have a direct implementation of the approach (this is still ongoing). For the time being, one can construct CMPs indirectly by constructing a c-net unfolding and compress it according to Definition 3.14. This allowed us to collect some data on the size of the CMPs that we can expect. In [RSK13], Section 5, we report the sizes of unfoldings and merged processes of c-nets, their plain encodings, and their PR-encodings. Over a standard set of benchmarks, CMPs were consistently the most compact among these six data structures. However, the merged processes of the plain encoding were not much bigger in all cases except one. The savings over unfoldings are quite significant, however, in many cases.

We carried out a case study on the mutual exclusion algorithm by Dijkstra [Dij65], which we modelled as a c-net. This model is quite interesting for us because it exemplifies both effects that we are trying to handle with CMPs:

Table 3.1: Unfolding and MP sizes of DIJKSTRA with n threads.

Net		Merged Processes			Unfoldings		
n	$\|T\|$	Ctx	Plain	PR	Ctx	Plain	PR
2	18	31	42	40	35	54	54
3	36	64	113	121	131	371	364
4	60	105	220	278	406	2080	1998
5	90	155	375	582	1139	10463	9822
6	126	214	589	1198	3000	49331	44993
	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(3^n)$	$\mathcal{O}(5^n)$	$\mathcal{O}(5^n)$

to enter the critical section, a thread needs to repeatedly check the variables of the other threads, and in fact multiple processes may check the same variable at the same time (concurrent read access). Moreover, the threads can make some non-deterministic choices. More details are given in [RSK13], Section 5.2. Table 3.1 reproduces the table of results, showing the number of (mp-)events in merged processes/unfoldings on the basis of c-nets and their two encodings, respectively. The growth of the CMP is in the order of $\mathcal{O}(n^2)$, where n ist the number of processes; indeed the size of the net itself is quadratic, so the behaviour of the CMP is linear w.r.t. the net. The other merged processes are also of polynomial size, approximately $\mathcal{O}(n^3)$. The different unfolding types are all of exponential size here.

3.3 Other contributions

This section gives brief descriptions of some other research. They concern the unfoldings of Petri nets, i.e. without read arcs. For these, I wrote a tool called MOLE, based on earlier work by Stefan Römer, which produces a finite complete prefix of a safe Petri net. It was the basis for some of the experimentation mentioned in the following subsections.

Let $\mathcal{N} = \langle P, T, F, m_0 \rangle$ be a Petri net. (We omit the context-relation C, which is $\emptyset$.) In this case, some definitions become slightly simplified. The following facts are well-known (see, e.g., [EH08]): Consider Definition 3.2. If there are no read arcs, then the causality relation $<$ becomes just the transitive closure of F.

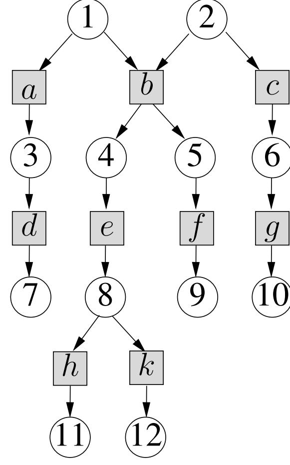

Figure 3.17: An occurrence net illustrating the reveals relation.

Let $\mathcal{N} = \langle B, E, G, \widehat{m}_0 \rangle$ be an occurrence net. In Definition 3.6, $e \lessdot e'$ iff $e^\bullet \cap {}^\bullet e'$, and the relation $\nearrow\!\!\!\nearrow$ is empty. Let x, x' be conditions or events. We define $x \# x'$ if there exist events $e \le x$ and $e' \le x'$ such that $e \#_i e'$, and $x \parallel y$ iff neither $x < y$, nor $x > y$, nor $x \# y$. Let $m = \{c_1, \ldots, c_n\}$ be a set of conditions. If $c \parallel c'$ holds for all pairs $c, c' \in m$, then $conc(m)$ holds (in contrast to c-nets, recall Figure 3.9). Finally, a configuration $\mathcal{C}$ (Definition 3.7) now becomes simply a finite, causally closed set of events, such that $\neg(e \# e')$ for any pair $e, e' \in \mathcal{C}$. The 'evolves' relation between configurations reduces to a subset relation, i.e. $\mathcal{C} \sqsubseteq \mathcal{C}'$ iff $\mathcal{C} \subseteq \mathcal{C}'$.

3.3.1 The reveals relation

This section contains a short summary of work on the *reveals* relation with Stefan Haar and Christian Kern [HKS11, HKS13]. Given an occurrence net $\mathcal{N} = \langle B, E, G, \widehat{m}_0 \rangle$, we study the following relation between events a, b: event a is said to *reveal* event b if, whenever a occurs, the occurrence of b is inevitable, be it before, after, or concurrently to a. Thus, the reveals relation does not generally imply a causal relation.

In this context, we consider executions that satisfy a criterion of *weak fairness*, i.e., an event that is enabled forever must eventually fire. E.g., in Figure 3.17, if ever there is a token on condition 3, then d must fire eventually because there is no other event competing for the token on 3. Under this

interpretation, if we write $a \triangleright b$ for "a reveals b", then we have the following examples in Figure 3.17:

- $a \triangleright d$ (see above); $d \triangleright a$;
- $e \triangleright f$ (because $e \triangleright b$ and $b \triangleright f$);
- $h \triangleright f$, but $\neg(f \triangleright h)$;
- $a \triangleright c$, and $c \triangleright a$.

The weak-fairness semantics is rather natural for occurrence nets – a weakly fair execution corresponds to a *maximal* configuration of an occurrence net, be it finite or infinite. This gives rise to a simple characterization of the reveals relation in terms of conflicts. For an event x, let $\#[x] := \{ y \mid x \# y \}$ denote all the events that are in (symmetric) conflict with x. In a maximal run, if none of the events in $\#[x]$ occurs, then x must eventually occur. Thus, we obtain the following characterization for two events x, y (see [Haa10]):

$$x \triangleright y \qquad \text{if and only if} \qquad \#[x] \supseteq \#[y]$$

In other words, x reveals y if the occurrence of x rules out all the events that could prevent y. On the other hand, if we can find an event that is in conflict with y but not x, i.e. some z with $\neg(x \# z)$ and $y \# z$, then $x \triangleright y$ does not hold, and we call z a *witness* for x, y.

The reveals relation was introduced by Haar in [Haa07, Haa10]. Its motivation lies in the study of systems that are partially observable. Suppose, for example, that we are observing the visible part of some ongoing execution of a Petri net. By exploiting structural properties of unfoldings and their explicit notions of causality, conflict, etc, we can often deduce from one observed event e that another event e' must be part of the current run, sometimes even if e' is in the future of e, in a different component, or causally unrelated.

We now summarize the main results. The first proposition corresponds to Theorem 2 in [HKS13]. It says that in an unfolding $\mathcal{U}_\mathcal{N}$ of a safe Petri net one can decide $x \triangleright y$ by inspecting a finite prefix of $\mathcal{U}_\mathcal{N}$. In the following, we call *height* of an event x (written $\mathcal{H}(x)$) the length of the longest chain of causally related events ending at x. The value $K(m)$ means the maximal height of an event in a marking-complete prefix for net $\mathcal{N}$ with m as initial marking.

Proposition 3.4 *Let $\mathcal{N} = \langle P, T, F, m_0 \rangle$ be a safe Petri net, let its unfolding be $\mathcal{U}_\mathcal{N} = \langle B, E, G, \widehat{m}_0 \rangle$, and let $K := \max\{ K(m) \mid m \subseteq P \}$. For any two events x, y such that $\neg(x \rhd y)$, there exists an event z such that*

(1) $\neg(x \# z)$ and $y \# z$;

(2) $\mathcal{H}(z) \leq n + K - 1$, where $n := \max(\mathcal{H}(x), \mathcal{H}(y))$.

Notes on the proof: The idea of the proof is relatively simple, but the details are not quite trivial. We show that if there exists a witness z whose height is larger than anticipated in (2), then we can perform some 'surgery' on the unfolding to obtain another witness with smaller height. Proposition 3.4 considerably improves the previous bound, given in [Haa10], for the size of the finite prefix needed to decide whether x reveals y. While the previous bound seemed to make this decision impracticable, the new bound gives much more hope to determine the relation in practice. In [HKS13] we give a class of examples showing that the new bound is tight.

The following proposition corresponds to Theorem 3 in [HKS13].

Proposition 3.5 *Given a safe Petri net $\mathcal{N}$ and two events x, y of its unfolding, it is PSPACE-complete do decide whether $x \rhd y$ holds.*

Notes on the proof: The hardness proof is by a simple reduction from the coverability problem. The technique for the upper bound is more interesting: we assume that the cones $[x]$ and $[y]$ are given. Having excluded some trivial cases (like $y < x$), we then try to find a witness for x, y. Thanks to Proposition 3.4 we can bound the depth of our search. However, we cannot simply construct the unfolding up to that height, which would not be possible in polynomial space. Instead, we non-deterministically simulate a run of the unfolding, remembering only the current cut. Part of the cut can be inside the given part $[x] \cup [y]$, other conditions in the cut may be 'beyond' $[x] \cup [y]$, and of these we just need to track their height.

The third main result in [HKS13] is an algorithm that computes the entire reveals relation on a given, finite occurrence net. The algorithm can be implemented completely with bitset operations and consists of three passes over the prefix; in the first, one computes the causal relation, in the second the conflict relation, and finally in the third the reveals relation. The algorithm takes cubic time w.r.t. the size of the prefix.

3.3.2 Weak diagnosis

Continuing in the vein of partially observable systems with weakly fair semantics, this section reports on a method for diagnosis, resulting from joint work with Stefan Haar and César Rodríguez, published in [HRS13].

Fault diagnosis is a classical problem in runtime analysis of systems. We are given a system, e.g., an LTS, whose possible behaviours we know precisely. However, its executions are only partially observable. In a typical automata-theoretic setting, the actions of the LTS are partitioned into *observable* and *unobservable* actions. The latter typically contains a special action called *fault*. Classically, the task of the observer is to determine, given a sequence of observable actions, whether a fault has occurred in that sequence.

Fault diagnosis has been studied in many variants. Here we study it for the setting where the LTS is given in form of a safe Petri net. We also slightly alter the goal: Instead of asking whether a fault has happened in the past, we wish to know whether all weakly fair runs that are compatible with our observations contain a fault, even if the fault has not yet occured but will inevitably do so in the future.

Having a concurrent model also opens the possibility for different setups of sensors and supervisors. In our setting we allow a multitude of sensors that report their observations to a single supervisor. Technically, we allow the observations to form a labelled partial order.

Asynchronous diagnosis of safe Petri nets with an unfolding-based approach has previously been presented by [BFHJ03] and [EK12]. Both use Petri net unfoldings under certain restrictions: [BFHJ03] accepts partial-order observations, but refuses models with unobservable loops; [EK12] accepts the latter, thanks to dedicated *cutoff criteria*, but refuses the former. Our work uses both features, additionally accounting for weak fairness in the diagnosis procedure.

In Section 3.3.1, the notion that some event has become inevitable having seen another is expressed through a binary relation between events. For diagnosis purposes, this binary relation is conceptually replaced by a more general notion of *extended reveals* [BCH11].

Without going into the technical details of the paper, we discuss the most important notions. The observations made about the system are represented as a labelled partial order α. The actual behaviour of the system is represented by the unfolding of the underlying safe net. We say that a configuration $\mathcal{C}$ of that unfolding *explains* α if the LPO of α can be embedded

into the causal structure of $\mathcal{C}$. Then we ask whether all maximal (i.e. weakly fair) runs containing such an explanation also contain a fault event (equation (6) in [HRS13]).

It is more convenient to formulate (6) in negated and hence existential form: Does there exist an explanation of our observation that can be continued forever with a fault-free run? Before we solve this problem, we have to overcome two obstacles: (i) an observation pattern α can have infinitely many explanations due to unobservable loops; (ii) an explanation has, in general, infinitely many continuations that must be handled finitely. Both problems are treated separately using two different cutoff criteria. While the technique for (i) is similar to [EK12], i.e. a restriction to so-called *succinct* explanations, the cutoff technique for (ii) is novel. Having reduced diagnosis to an existential, finite problem, we discuss how to solve it by computing adequate unfolding prefixes and encoding the entire problem into SAT. An actual implementation of the approach does not yet exist, and would still require some engineering effort so resolve certain practical issues, which we detail in the paper.

3.3.3 A note on depth-first-search order

Algorithm 2 is parametrized by an ordering $\prec$ between configurations that determines the order in which (enriched) events are added to the prefix. When one produces a finite marking-complete prefix, then $\prec$ influences the precise shape of the outcome. More interestingly, it can also determine whether the result is really a marking-complete prefix or not!

McMillan [McM92] originally showed that the order $\prec_M$ defined by $\mathcal{C}_1 \prec_M \mathcal{C}_2$ iff $|\mathcal{C}_1| < |\mathcal{C}_2|$, is correct, i.e. leads to a marking-complete prefix. This order is partial, i.e., it is not determined for configurations of the same size. As Esparza et al [ERV02] showed, this may lead to a marking-complete prefix larger than the reachability graph itself. However, if one replaces $\prec_M$ by an order that is *total* and *adequate*, then the algorithm terminates with a complete prefix, and they propose one such order, which we call $\prec_{ERV}$. The precise details of $\prec_{ERV}$ not important at this point – what matters is that $\prec_{ERV}$ refines $\prec_M$, i.e. compares by size first. In a certain sense, this means that the construction proceeds in breadth-first manner: first all events e with size $|[e]| = 1$, then with size 2, 3, etc.

Adequate orders are known to be correct, but sometimes other search strategies would be desirable, e.g., when the solution is known to lie at a

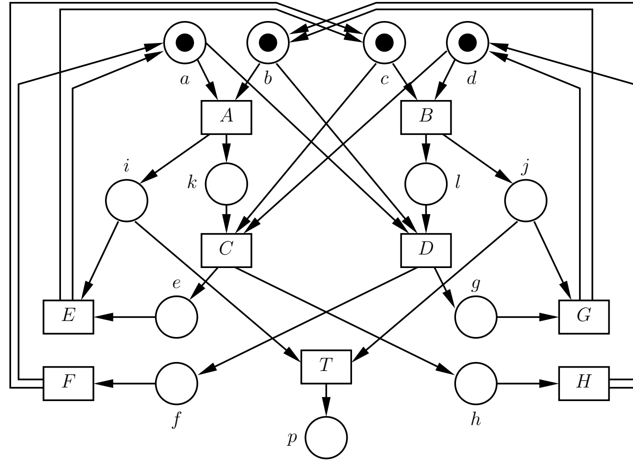

Figure 3.18: Counterexample net for depth-first unfolding.

certain depth in the unfolding. Unfortunately, as a short contribution by Javier Esparza, Pradeep Kanade, and myself [EKS08a] shows, depth-first search (DFS) may lead to incorrect results. More precisely, let us call *depth-first* an ordering that has the following property: If $\mathcal{P}$ is the current prefix and e the latest event added to $\mathcal{P}$, then the possible extensions enabled by e will be explored before any other possible extension of $\mathcal{P}$. We shall give a small net that, when unfolded under any depth-first ordering, misses at least one marking in the finished prefix.

The net is shown in Figure 3.18. Without going into all details, the reader can convince himself that:

- From the initial marking, the net can either execute A, B, T, ending with a token on place p; or A, C, E, H (back to the initial marking), or B, D, F, G (ditto).

- Depending on its inital choice, a depth-first unfolder will find either the loop A, C, E, H followed by (because the next A is now cutoff) the other loop B, D, F, G (or vice versa). After the second loop, the third instances of A, B are going to be cutoffs.

- Only at that point will depth-first search fall back to the initial B (resp. A) that was not considered in the beginning. But this, too, is now a cutoff because of the earlier B. So A, B are never executed in parallel inside the prefix, so T is never included in the final prefix.

3.3.4 Improved detection of possible extensions

In Section 3.2.2, we discussed an efficient algorihm for constructing a c-net unfolding. Part of this effort was the development of a concurrency relation between enriched conditions. Keeping this information in memory, i.e. whether $\rho \parallel \rho'$ for all pairs of enriched conditions, allows to determine possible extensions very quickly. For instance, CUNF uses this trick.

An analogous technique, but with a concurrency relation between conditions (i.e., without histories) is used in tools for Petri nets without read arcs such as MOLE or PUNF. They keep the concurrency relation $\parallel$ in memory, updating it whenever new conditions are added to the unfolding. On the other hand, the concurrency relation often dominates the memory footprint of the unfolder, since it is quadratic in the number of events.

For this reason, the tool PUNF (by Victor Khomenko) has an option to drop the concurrency relation from memory and determine such queries by inspecting directly the structure of the prefix computed so far. It goes without saying that this is more time-consuming, so Khomenko and Koutny [KK01] made a proposal to soften the impact. Whenever we have added a new event $e := \langle X, t\rangle$, i.e. with preset $X \subseteq B$ and label $t \in T$, then we proceed in two steps

1. For each place $p \in {}^\bullet(t^{\bullet\bullet}) \setminus t^\bullet$, determine the set $C(p, e)$ of p-labelled conditions $\langle x, p\rangle$ that are concurrent with e. For $p \in t^\bullet$, we set $C(p, e) := \{\langle e, p\rangle\}$

2. For all $t' \in t^{\bullet\bullet}$, use the sets $C(p, e)$ to discover new possible extensions, i.e., find coverable subsets X' with $h(X') = t'$ and add these to the possible extensions.

The proposal by Khomenko and Koutny concerns the second step: sets $C(p, e)$ and combinations thereof are intelligently re-used when possible by building so-called *preset trees*.

A recent short paper with César Rodríguez concerned an improvement for step 1 instead [RS13]. It exploits structural properties of unfoldings more stringently than before. In a safe net, e.g., no two conditions labelled by the same place p are concurrent. Instead, they must be either causally related or in conflict. The causal relation $<$, restricted to these conditions, forms therefore a forest, which we call the p-forest. Within this forest, we can exploit certain structural laws, for instance: if $c \# e$ and $c < c'$, then

$c' \# e$, and hence $c' \notin C(p, e)$. We exploit several such laws to speed up the computation of $C(p, e)$ and report on experiments on standard benchmarks. These are quite encouraging: our new algorithm runs 3.5 times faster than PUNF (when the latter is run without remembering the concurrency relation). It also runs faster than MOLE (*with* concurrency relation) in half of the cases, using on average half as much memory. Over the set of all benchmarks, however, MOLE remains two times faster than the new algorithm. Also, further improvements to our algorithm are possible, e.g. the combination with preset trees.

Chapter 4

Conclusion

This document summarizes my research in verification and, more generally, formal methods. It covers a range of various topics: sequential and concurrent systems; finite and infinite state systems; techniques like symbolic model checking, counterexample-guided abstraction refinement, unfoldings, SAT; and various applications like verification, data-flow analysis, authorization, or diagnosis.

My main interest in all this has been in algorithmic aspects; to understand, exhibit, and put to good use basic principles of computation. For instance, the bounded WPDS framework is built on very simple principles, yet it allows to extract answers to complex questions and has found many applications. For c-net unfoldings, which seemed much more complicated to deal with than the traditional case, we finally managed to find a single binary relation, called asymmetric concurrency, describing the solution to the main algorithmic problem for an efficient implementation. I am also pleased to have found a concrete use for my work on authorization systems inside our own teaching department, to name another example.

Pushdown model checking has become a vast field during the last decade, to the point where it is becoming difficult to keep track of the developments, the relationship between the different models, and the overarching principles behind them. By comparison, unfolding techniques have been getting less attention by the verification community.

Another technique that deals with state-space explosion due to concurrency is *partial-order reduction*. This technique has been highly successful due to a large amount of algorithmic engineering, notably embodied in the tool Spin. Work on the unfolding approach has focused on Petri nets, and

while there exist efficient tools for that, Petri nets are a relatively general low-level formalism, so these techniques do not exploit properties of higher language features. By contrast, Spin comes with its own modeling language having an explicit notion of process, communication channels, and variables. Indeed, the reduction techniques implemented in Spin exploit the specific properties of these features. This would seem to be a viable perspective for unfoldings, too. For instance, towards the end of [RSK13], we sketched how locks could be handled in unfoldings if one integrated them directly into the formalism. So an interesting direction of research would be a generalization of unfoldings to higher-level formalisms.

A current direction of research that I would be interested in pursuing further is the recently begun work on active diagnosis [HHMS13]. Here, one is given a partially observable system and aims at controlling it in order to make it diagnosable. This is a natural and motivating problem with interesting connections to automata and game theory. We have only started the work on this subject, so there remain important open questions and perspectives for extension.

Acknowledgements It has been a long road, and this thesis would not have been possible without the goodwill, help, and patience of many people to whom I am grateful for all they have done. This applies in particular to my mentors, Javier Esparza and Stefan Haar, both of whom I cannot thank enough for their guidance throughout these years.

I am indebted to those who have introduced me to new topics and enlargened my horizons. With respect to this thesis, this applies in particular to Thomas Reps and Somesh Jha at the University of Madison-Wisconsin, who taught me about dataflow analysis and authorization; and to Paolo Baldan, for introducing me to the subject to contextual nets.

Working in two countries, Germany and France, and also in two different types of laboratories, has been an enriching experience in many ways. The close personal ties in the comparatively small working groups in Stuttgart and Munich, and the wider horizons in a large institute like the LSV have been complementary and positive experiences. Both proved to be fruitful environments that provided me with all the help I could possibly ask for.

Sriram Rajamani invited me to his group in Bangalore on two occasions. Since the resulting work was not along the main lines of my research, it did not feature in this thesis, but this, too, was an immensely enriching

experience. I sincerely thank him, as well as Andreas Podelski, who trusted to stand in for him at the University of Freiburg during one semester.

If it is the apprentice that requires a chance to prove himself, the same is true of the teacher. César Rodríguez accepted my guidance as his PhD supervisor in a situation that carried many unknowns for him. I am humbled and grateful for the trust that he has conferred upon me in this way. The successful development of his own thesis, which is due to be defended soon, has helped to alleviate the self-doubts concerning my adequacy for a habilitation.

Several institutes and organizations have not only paid me but also provided office space and more importantly a sense of belonging. They are the University of Stuttgart, the Technical University of Munich, the University of Freiburg, and École Normale Supérieure de Cachan. At the latter, I profited from a *Chaire* of the French national institute INRIA. Moreover, I was a Visiting Researcher at the University of Wisconsin at Madison and Microsoft Research India. I thank all scientific colleagues, clerical and technical staff at those institutes for their friendship and support.

Finally, I cannot finish without giving the place of honour to my parents. It is not just because the world has changed in so many ways since my birth, but also, specifically, thanks to their decisions, their love and support, that I have had opportunities that they never knew. Thank you for all.

Bibliography

[ABT08] Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. On the reachability analysis of acyclic networks of pushdown systems. In *Proceedings of CONCUR*, LNCS 5201, pages 356–371, 2008.

[AEY01] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Analysis of recursive state machines. In *Proceedings of CAV*, LNCS 2102, pages 207–220, 2001.

[ALSU86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 1986.

[BBC+10] Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, and Stefan Schwoon. On the computation of McMillan's prefix for contextual nets and graph grammars. In *Proceedings of ICGT*, LNCS 6372, pages 91–106. Springer, 2010.

[BBC+12] Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient unfolding of contextual Petri nets. *Theoretical Computer Science*, 449(1):2–22, August 2012.

[BCH11] Sandie Balaguer, Thomas Chatain, and Stefan Haar. Building tight occurrence nets from reveals relations. In *Proceedings of ACSD*, pages 44–53. IEEE, 2011.

[BCHS12] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. A saturation method for collapsible pushdown systems. In *Proceedings of ICALP, Part II*, LNCS 7392, pages 165–176, 2012.

[BCKS07] Paolo Baldan, Andrea Corradini, Barbara König, and Stefan Schwoon. McMillan's complete prefix for contextual nets. In *Proceedings of the UFO workshop*, pages 32–49, June 2007.

[BCKS08] Paolo Baldan, Andrea Corradini, Barbara König, and Stefan Schwoon. McMillan's complete prefix for contextual nets. *Transactions on Petri Nets and Other Models of Concurrency*, 1:199–220, November 2008. LNCS 5100.

[BCM98] Paolo Baldan, Andrea Corradini, and Ugo Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. In *Proceeding of FoSSaCS*, LNCS 1378, pages 63–80, 1998.

[BCM01] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual Petri nets, asymmetric event structures, and processes. *Information and Computation*, 171(1):1–49, 2001.

[BE12] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. In *Proceedings of TACAS*, LNCS 7214, pages 451–465, 2012.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proceedings of CONCUR 1997*, LNCS 1243, pages 135–150, 1997.

[BESS05] Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček. Reachability analysis of multithreaded software with asynchronous communication. In *Proceedings of FSTTCS*, LNCS 3821, pages 348–359. Springer, December 2005.

[BESS08] Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Dejvuth Suwimonteerabuth. SDSIrep: A reputation system based on SDSI. In *Proceedings of TACAS*, LNCS 4963, pages 501–516. Springer, April 2008.

[BET03a] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal on Foundations of Computer Science*, 14(4):551–582, 2003.

[BET03b] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of POPL*, pages 62–73, 2003.

[BFHJ03] Albert Benveniste, Eric Fabre, Stefan Haar, and Claude Jard. Diagnosis of asynchronous discrete-event systems: a net unfolding approach. *IEEE Transactions on Automatic Control*, 48(5):714–727, 2003.

[BFQ07] Ahmed Bouajjani, Séverine Fratani, and Shaz Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *Proceedings of CAV*, LNCS 4590, pages 207–220, 2007.

[BGR01] Michael Benedikt, Patrice Godefroid, and Thomas W. Reps. Model checking of unrestricted hierarchical state machines. In *Proceedings of ICALP*, pages 652–666, 2001.

[BGRT05] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. CodeSurfer/x86 - a platform for analyzing x86 executables. In *Conference on Compiler Construction*, LNCS 3443, pages 250–254, 2005.

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proceedings of CONCUR*, LNCS 3653, pages 473–487, 2005.

[BO93] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Texts and Monographs in Computer Science. Springer-Verlag, Heidelberg, 1993.

[BR00] Tom Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the SPIN Workshop*, LNCS 1885, pages 113–130, 2000.

[BR01] Tom Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN*, LNCS 2057, pages 103–122, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS57] Friedrich Ludwig Bauer and Klaus Samelson. Verfahren zur automatischen verarbeitung von kodierten daten und rechenmaschine zur ausbung des verfahrens, 1957. Patent application.

[Büc64] Julius Richard Büchi. Regular canonical systems and finite automata. *Arch. Math. Logik*, 6:91–111, 1964.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proceedings of ICSE*, pages 385–395. IEEE, 2003.

[CCK+06] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Verifying concurrent message-passing C programs with recursive calls. In *Proceedings of TACAS*, LNCS 3920, pages 334–349, 2006.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV*, LNCS 1855, pages 154–169, 2000.

[CGS09] Michael Codish, Samir Genaim, and Peter J. Stuckey. A declarative encoding of telecommunications feature subscription in SAT. In *Proceedings of PPDP*, pages 255–266. ACM, 2009.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569ff, September 1965.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylönen. *RFC 2693: SPKI Certificate Theory*. The Internet Society, September 1999.

[EGS05] Javier Esparza, Pierre Ganty, and Stefan Schwoon. Locality-based abstractions. In *Proceedings of SAS*, LNCS 3672, pages 118–134. Springer, September 2005.

[EH01] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In *Proceedings of SPIN*, LNCS 2057, pages 37–56, 2001.

[EH08] Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV*, LNCS 1855, pages 232–247. Springer, July 2000.

[EJS13] Javier Esparza, Loïg Jezequel, and Stefan Schwoon. Computation of summaries using net unfoldings. In *Proceedings of FSTTCS*, December 2013. To appear.

[EK99] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proceedings of FoSSaCS*, LNCS 1578, pages 14–30, 1999.

[EK12] Javier Esparza and Christian Kern. Reactive and proactive diagnosis of distributed systems using net unfoldings. In *Proceedings of ACSD*, pages 154–163, 2012.

[EKL07a] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. An extension of Newton's method to ω-continuous semirings. In *Proceedings of DLT*, volume 4588 of *LNCS 4588*, pages 157–168, 2007.

[EKL07b] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. On fixed point equations over commutative semirings. In *Proceedings of STACS*, LNCS 4393, pages 296–307, 2007.

[EKL08] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newton's method for ω-continuous semirings. In Luca Aceto, Magnus M. Halldorsson, and Anna Ingolfsdottir, editors, *Proceedings of ICALP*, LNCS 5126, pages 14–26, 2008. Invited paper.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proceedings of TACAS*, LNCS 3920, pages 489–503. Springer, 2006.

[EKS08a] Javier Esparza, Pradeep Kanade, and Stefan Schwoon. A negative result on depth-first unfoldings. *Software Tools for Technology Transfer*, 10(2):161–166, 2008.

[EKS08b] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:27–56, June 2008. Special Issue on Constraints to Formal Verification.

[ERS00] Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *EATCS Bulletin*, 72:169–177, October 2000.

[ERV02] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

[ES01] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Proceedings of CAV*, LNCS 2102, pages 324–336, July 2001.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of SAT*, LNCS 2919, pages 502–518, 2003.

[For56] Lester R. Ford. Network flow theory. Technical Report P-923, RAND Corporation, August 1956.

[FWW97] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[Gai08] Andreas Gaiser. Erreichbarkeitsanalyse funktionaler programme mit grundtermersetzungssystemen. Master's thesis, Universität Stuttgart, 2008. In German.

[GS07a] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *Proceedings of ESOP*, LNCS 4421, pages 300–315, 2007.

[GS07b] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *Proceedings of CSL*, LNCS 4646, pages 23–40, 2007.

[GS09] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. In *Proceedings of the MEMICS workshop*, pages 69–77, November 2009.

[Haa07] Stefan Haar. Unfold and cover: Qualitative diagnosability for Petri nets. In *Proceedings of CDC*, pages 1886–1891. IEEE, 2007.

[Haa10] Stefan Haar. Types of asynchronous diagnosability and the reveals-relation in occurrence nets. *IEEE Transactions on Automatic Control*, 55(10):2310–2320, 2010.

[Ham57] Charles Leonard Hamblin. Computer languages. *The Australian Journal of Science*, 20:135–139, 1957.

[Hel99] Keijo Heljanko. *Deadlock and Reachability Checking with Finite Complete Prefixes*. Licentiate's thesis, Helsinki University of Technology, 1999.

[HHMS13] Stefan Haar, Serge Haddad, Tarek Melliti, and Stefan Schwoon. Optimal constructions for active diagnosis. In *Proceedings of FSTTCS*, December 2013. To appear.

[HJMS02] Tom A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of POPL*, pages 58–70. ACM Press, 2002.

[HK99] Mark W. Hopkins and Dexter Kozen. Parikh's theorem in commutative Kleene algebra. In *Proceedings of LICS*, pages 394–401, 1999.

[HKS11] Stefan Haar, Christian Kern, and Stefan Schwoon. Computing the reveals relation in occurrence nets. In *Proceedings of GandALF*, volume 54 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–44, June 2011.

[HKS13] Stefan Haar, Christian Kern, and Stefan Schwoon. Computing the reveals relation in occurrence nets. *Theoretical Computer Science*, 493:66–79, July 2013.

[HLMS10] Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. Reachability analysis of communicating pushdown systems. In *Proceedings of FOSSACS*, LNCS 6014, pages 267–281, 2010.

[HO07] Matthew Hague and C.-H. Luke Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In *Proceedings of FOSSACS*, LNCS 4423, pages 213–227, 2007.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HRS13] Stefan Haar, César Rodríguez, and Stefan Schwoon. Reveal your faults: It's only fair! In *Proceedings of ACSD*, pages 127–136, July 2013.

[HS04a] Markus Holzer and Stefan Schwoon. Assembling molecules in Atomix is hard. *Theoretical Computer Science*, 303(3):447–462, February 2004.

[HS04b] Markus Holzer and Stefan Schwoon. Reflections on Reflexion – computational complexity considerations on a puzzle game. In *Proceedings of the Third International Conference on FUN with Algorithms*, pages 90–105. Università di Pisa, May 2004.

[JK91] Ryszard Janicki and Maciej Koutny. Invariant semantics of nets with inhibitor arcs. In *Proceedings of CONCUR*, LNCS 527, pages 317–331, 1991.

[JR02] Somesh Jha and Thomas Reps. Analysis of SPKI/SDSI certificates using model checking. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 129–146. IEEE Computer Society, June 2002.

[JSWR06] Somesh Jha, Stefan Schwoon, Hao Wang, and Thomas Reps. Weighted pushdown systems and trust-management systems. In *Proceedings of TACAS*, LNCS 3920, pages 1–26. Springer, 2006. Invited paper.

[Kah09] Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of CFL-reachability for threads communicating via locks. In *Proceedings of LICS*, pages 27–36. IEEE, 2009.

[Kho] Victor Khomenko. The Punf unfolding tool. Available at http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/.

[KIG05] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of CAV*, LNCS 3576, pages 505–518, 2005.

[KK00] Victor Khomenko and Maciej Koutny. LP deadlock checking using partial order dependencies. In *Proc. CONCUR*, LNCS 1877, pages 410–425, 2000.

[KK01] Victor Khomenko and Maciej Koutny. Towards an efficient algorithm for unfolding Petri nets. In *Proc. CONCUR*, LNCS 2154, pages 366–380, 2001.

[KK07] Victor Khomenko and Maciej Koutny. Verification of bounded Petri nets using integer programming. *Formal Methods in System Design*, 30(2):143–176, 2007.

[KKKV06] Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. Merged processes – a new condensed representation of Petri net behaviour. *Acta Informatica*, 43(5):307–330, 2006.

[KKV03] Victor Khomenko, Maciej Koutny, and Walter Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40(2):95–118, 2003.

[KM11] Victor Khomenko and Andrey Mokhov. An algorithm for direct construction of complete merged processes. In *Proceedings of Petri Nets*, LNCS 6709, pages 89–108, 2011.

[KRML06] Nicholas Kidd, Thomas Reps, David Melski, and Akash Lal. WALi: the weighted automata library, 2006. http://research.cs.wisc.edu/wpis/wpds/index.php.

[KSSK09a] Morten Kühnrich, Stefan Schwoon, Jiří Srba, and Stefan Kiefer. Interprocedural dataflow analysis over weight domains with infinite descending chains. In *Proceedings of FOSSACS*, LNCS 5504, pages 440–455. Springer, March 2009.

[KSSK09b] Morten Kühnrich, Stefan Schwoon, Jiří Srba, and Stefan Kiefer. Interprocedural dataflow analysis over weight domains with infinite descending chains. Technical Report 0901.0501, Computing Research Repository, 2009.

[LMP08] Salvatore La Torre, Parthasaray Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In *Proceedings of TACAS*, LNCS 4963, pages 299–314, 2008.

[LMP09] Salvatore La Torre, Parthasaray Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *Proceedings of CAV*, LNCS 5643, pages 477–492, 2009.

[LN11] Salvatore La Torre and Margherita Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *Proceedings of CONCUR*, LNCS 6901, pages 203–218, 2011.

[Löd06] Christof Löding. Reachability problems on regular ground tree rewriting graphs. *Theory of Computing Systems*, 39(2):347–383, 2006.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LRB05] Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. In *Proceedings of CAV*, LNCS 3576, pages 434–448, 2005.

[LS07] Jérôme Leroux and Grégoire Sutre. Accelerated data-flow analysis. In *Proceedings of SAS*, LNCS 4634, pages 184–199, 2007.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Proceedings of TACAS*, LNCS 4963, pages 282–298, 2008.

[May98] Richard Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU München, 1998.

[McM92] Ken L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[McM03] Ken L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of CAV*, LNCS 2725, pages 1–13, 2003.

[MOS04] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *Proceedings of POPL*, pages 330–341, 2004.

[MR94] Ugo Montanari and Francesca Rossi. Contextual occurrence nets and concurrent constraint programming. In *Dagstuhl Seminar 9301*, LNCS 776, pages 280–295, 1994.

[MR95] Ugo Montanari and Francesca Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995.

[MR97] Stefan Melzer and Stefan Römer. Deadlock checking using net unfoldings. In *Proceedings of CAV*, LNCS 1254, pages 352–363, 1997.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13:85–108, 1981.

[NSRL06] Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. NETRA: Seeing through access control. In *Proceedings of FMSE*, pages 55–66, November 2006.

[NSS59] Allen Newell, John Clifford Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *International Conference on Information Processing*, pages 256–264, 1959.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS 2005*, LNCS 3440, pages 93–107, 2005.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *Proceedings of PLDI*, pages 14–24. ACM, 2004.

[Ram96] Ganesan Ramalingam. *Bounded Incremental Computation*. LNCS 1089. Springer, 1996.

[Ram00] Ganesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2):416–430, 2000.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998.

[Ris94] Gioia Ristori. *Modelling Systems with Shared Resources via Petri Nets*. PhD thesis, Department of Computer Science, University of Pisa, 1994.

[Rod] César Rodríguez. Cunf. http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/.

[RS12] César Rodríguez and Stefan Schwoon. Verification of Petri nets with read arcs. In *Proceedings of Concur*, LNCS 7454, pages 471–485. Springer, September 2012.

[RS13] César Rodríguez and Stefan Schwoon. An improved construction of Petri net unfoldings. In *Proceedings of the FSFMA workshop*, volume 31 of *OASICS*, pages 47–52. Schloss Dagstuhl, July 2013.

[RSB11] César Rodríguez, Stefan Schwoon, and Paolo Baldan. Efficient contextual unfolding. In *Proceedings of Concur*, LNCS 6901, pages 342–357. Springer, September 2011.

[RSJ03] Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of SAS*, LNCS 2694, pages 189–213. Springer, June 2003.

[RSJM05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, October 2005. Special Issue on the Static Analysis Symposium 2003.

[RSK13] César Rodríguez, Stefan Schwoon, and Victor Khomenko. Contextual merged processes. In *Proceedings of ICATPN*, LNCS 7927, pages 29–48. Springer, June 2013.

[SBSE07] Dejvuth Suwimonteerabuth, Felix Berger, Stefan Schwoon, and Javier Esparza. jMoped: A test environment for Java programs. In *Proceedings of CAV*, LNCS 4590, pages 164–167. Springer, July 2007.

[Scha] Stefan Schwoon. Moped – a model-checker for pushdown systems. http://www.lsv.ens-cachan.fr/~schwoon/tools/moped/.

[Schb] Stefan Schwoon. Mole. http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/.

[Schc] Stefan Schwoon. WPDS: A library for weighted pushdown systems. http://www.lsv.ens-cachan.fr/~schwoon/tools/wpds/.

[Sch02a] Stefan Schwoon. Determinization and complementation of Streett automata. In *Automata, Logics, and Infinite Games*, LNCS 2500, chapter 5, pages 79–91. Springer, 2002.

[Sch02b] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In *Proceedings of TACAS*, LNCS 3440, pages 174–190. Springer, April 2005.

[SES08] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *Proceedings of SPIN*, LNCS 5156, pages 270–287. Springer, 2008.

[SJRS03] Stefan Schwoon, Somesh Jha, Thomas Reps, and Stuart Stubblebine. On generalized authorization problems. In *Proceedings of CSFW*, pages 202–218. IEEE Computer Society, June 2003.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[SS08] Martin Sachenbacher and Stefan Schwoon. Model-based test generation using quantified CSPs: A map. In *Proceedings of the ECAI 2008 Workshop on Model-Based Systems*, pages 37–41, July 2008.

[SSE05] Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jMoped: A Java bytecode checker based on Moped. In *Proceedings of TACAS*, LNCS 3440, pages 541–545. Springer, April 2005. Tool paper.

[SSE06] Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *Proceedings of ATVA*, LNCS 4218, pages 141–153. Springer, October 2006.

[Suw09] Dejvuth Suwimonteerabuth. *Reachability in Pushdown Systems: Algorithms and Applications*. PhD thesis, Technische Universität München, 2009.

[TM07] Akihiko Tozawa and Yasuhiko Minamide. Complexity results on balanced context-free languages. In *Proceedings of FoSSaCS*, LNCS 4423, pages 346–360, 2007.

[Tur45] Alan M. Turing. Proposal for the development in the mathematics division of an automatic computing engine. Technical Report E882, National Physical Laboratory, 1945.

[VSY98] Walter Vogler, Alexei L. Semenov, and Alexandre Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proceedings of CONCUR*, LNCS 1466, pages 501–516, 1998.

[VW86] Moshe Y. Vardi and Pierre Wolper. Automata theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[Wal96] Igor Walukiewicz. Pushdown processes: Games and model checking. In *Proceedings of CAV*, LLNCS 1102, pages 62–74, 1996.

[Wal00] Igor Walukiewicz. Model checking CTL properties of pushdown systems. In *Proceedings of FSTTCS*, LNCS 1974, pages 127–138, 2000.

[Wen10] Alexander Wenner. Weighted dynamic pushdown networks. In *Proceedings of ESOP*, LNCS 6012, pages 590–609, 2010.

[Win02] Józef Winkowski. Reachability in contextual nets. *Fundamenta Informaticae*, 51(1–2):235–250, 2002.

[WJR+06] Hao Wang, Somesh Jha, Thomas Reps, Stefan Schwoon, and Stuart Stubblebine. Reducing the dependence of SPKI/SDSI on PKI.

In *Proceedings of ESORICS*, LNCS 4189, pages 156–173. Springer, September 2006.

[WL93] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *Research in Security and Privacy*, pages 178–194. IEEE, 1993.