University of Newcastle upon Tyne
School of Computing Science

Model Checking Based on Prefixes of Petri Net Unfoldings

by
Victor Khomenko

PhD Thesis

February 2003

Contents

Acknowledgements **iv**

Abstract **v**

Introduction **vi**

1 Basic Notions **1**

1.1 Multisets 1
1.2 Noetherian induction 2
1.3 Petri nets 2
1.4 Marking equation 4
1.5 Branching Processes 5

2 Canonical Prefixes **10**

2.1 Semantical meaning of completeness 10
2.2 Algorithmics of prefix generation 11
2.3 The new approach 12
2.4 König's Lemma for branching processes 13
2.5 Complete prefixes of Petri net unfoldings 14
2.5.1 Cutting context 14
2.5.2 Completeness of branching processes 16
2.6 Canonical prefix 18
2.6.1 Static cut-off events 18
2.6.2 Canonical prefix and its properties 19
2.7 Unfolding algorithms 23
2.7.1 ERV unfolding algorithm 23
2.7.2 Unfolding with slices 24
2.8 Conclusions 27

3 Test Bench **28**

3.1 Testing Unfolding Algorithms 28
3.2 Test cases 29

4 Computing Possible Extensions **33**
4.1 Employing the UpdatePotExt function 34
4.2 Reducing the number of candidates 35
4.3 Preset trees 38
4.4 Building preset trees 39
4.5 Implementation issues 43
4.6 Experimental results 48
4.7 Conclusions 51

5 Parallel Unfolding Algorithm **53**
5.1 The slicing algorithm 53
5.2 Choosing a slice 54
5.3 Cut-offs 'in advance' 55
5.4 Parallelizing the unfolding algorithm 56
5.5 Implementation issues 58
5.5.1 Minimizing interlocking 58
5.5.2 Computing final markings 59
5.5.3 Optimizing computation of possible extensions 60
5.6 Experimental results 61
5.7 Conclusions 62

6 Unfoldings of High-Level Petri Nets **64**
6.1 High-level Petri nets 65
6.1.1 M-nets 65
6.1.2 M-net systems 66
6.2 Translation into low-level nets 68
6.3 Branching processes of high-level Petri nets 69
6.4 M-net unfolding algorithm 70
6.5 Case studies 71
6.5.1 Greatest common divisor 72
6.5.2 Mutual exclusion algorithm 73
6.6 Conclusions 76

7 Prefix-Based Model Checking **77**
7.1 Deadlock detection using linear programming 77
7.2 Solving systems of linear constraints 79
7.3 Integer programming verification algorithm 84
7.3.1 Reduction to a pure integer problem 84
7.3.2 Partial-order dependencies between variables 85
7.3.3 Compatible closures 86
7.3.4 Removal of redundant constraints 87
7.3.5 Extending CDA (intuition) 88
7.3.6 Developing an extension of CDA 88
7.3.7 Applying the method for Σ^{Θ}-compatible vectors 94

7.4 Implementation of the algorithm . . . 94
7.4.1 Retrieving a solution . . . 96
7.4.2 Shortest trail . . . 96
7.5 Optimizations . . . 97
7.6 Extended reachability analysis . . . 99
7.6.1 Deadlock checking in safe case . . . 100
7.6.2 Terminal markings . . . 101
7.7 Other verification problems . . . 102
7.7.1 Mutual exclusion . . . 102
7.7.2 Reachability and coverability . . . 103
7.8 Further optimization for deadlock detection . . . 103
7.9 On-the-fly deadlock detection . . . 104
7.10 Efficiency of the branching condition . . . 105
7.11 Parallelization issues . . . 106
7.12 Experimental results . . . 108
7.13 Conclusions . . . 109

8 Detecting State Coding Conflicts in STGs 112
8.1 Basic definitions . . . 114
8.1.1 Signal Transition Graphs . . . 114
8.1.2 STG branching processes . . . 116
8.2 State coding conflict detection using integer programming . . . 117
8.2.1 Encoding constraint . . . 118
8.2.2 USC separating constraint . . . 119
8.2.3 CSC separating constraint . . . 119
8.2.4 Retrieving a solution . . . 120
8.3 Verifying the normalcy property . . . 120
8.4 The case of conflict-free nets . . . 122
8.5 Handling dummy events . . . 123
8.6 Experimental results . . . 125
8.7 Conclusions . . . 128

Conclusions 130

Bibliography 132

Index 141

Acknowledgements

I would like to thank God for his constant help and inspiration during my study.

This work would not have been possible without intensive collaboration with many people. In particular, I would like to express my gratitude to my supervisors Maciej Koutny and Paul Watson in Newcastle University, and my former supervisor Alexander Letichevsky in Glushkov Institute of Cybernetics for initiating this research and for all the help and moral support they provided.

Most of the ideas described in this thesis are a result of numerous discussions with Alex Bystrov, Javier Esparza, Keijo Heljanko, Sergei Krivoi, Christian Stehno, Walter Vogler and Alex Yakovlev, and, of course, my supervisors. In addition, I would like to thank Jordi Cortadella for compiling a special version of the PETRIFY tool used for the experiments in Chapter 8, Stephan Melzer for sending his PhD thesis, Peter Rossmanith for suggesting the proof of an important proposition in Chapter 4, and Jason Steggles for many helpful comments.

An essential contributing factor to this research was support of my family, which I received in abundance from my wife Oksana and my parents Vladimir and Polina, in spite of the fact that Petri nets are not their favourite topic of conversation.

This research was supported by the ORS Awards Scheme grant ORS/C20/4, the EPSRC grants GR/M99293 and GR/M94366 (MOVIE), the ACID-WG grant IST-1999-29119, and the ARC grant 1175 (JIP).

Abstract

The human society is becoming increasingly dependent on automated control systems, and the correct behaviour and reliability of the hardware and software used to implement them is often of a paramount importance. Yet, the growing complexity of such system makes it hard to design them without defects.

Especially difficult is the development of concurrent systems, because they are generally harder to understand, and errors in them often do not show up during the testing. Therefore, the conventional methods are not sufficient for establishing the correctness of such systems, and some kind of computer-aided *formal verification* is required.

One of the most popular formal verification techniques is *model checking*, in which the verification of a system is carried out using a finite representation of its state space. While being convenient in practice, it suffers a major drawback, viz. the *state space explosion* problem. That is, even a relatively small system specification can (and often does) yield a very large state space.

To alleviate this problem, a number of methods have been proposed. Among them, a prominent technique is McMillan's (finite prefixes of) Petri net unfoldings. It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net, called a *prefix*. Often such prefixes are exponentially smaller than the corresponding reachability graphs, especially if the system at hand exhibits a lot of concurrency.

This thesis is all about efficient verification based on this technique. It discusses the theory of finite and complete prefixes of Petri net unfoldings and provides several practical verification algorithms.

On one hand, the thesis addresses the issue of efficient prefix generation, suggesting a new method of computing possible extensions of a branching process, a parallel unfolding algorithm, and an unfolding algorithm for a class of high-level Petri nets. On the other hand, it shows how unfolding prefixes can be used for practical verification, proposing a new integer programming verification technique. This technique can be used to check many fundamental properties, such as deadlock freeness, mutual exclusion, reachability and coverability of a marking, and extended reachability. Moreover, it can be applied to check state encoding properties of specifications of asynchronous circuits, yielding a fast and memory-efficient algorithm.

Introduction

The human society is becoming increasingly dependent on automated control systems, and the correct behaviour and reliability of the hardware and software used to implement them is often of a paramount importance. A failure of such a safety-critical application as an air traffic control system or a nuclear reactor can entail a major casualty. Even when human lives are not involved, implementation mistakes can sometimes lead to substantial economic loss, e.g., the notorious floating point division bug in the Pentium microprocessor cost Intel $475 million (see, e.g., [27]).

Yet, the growing complexity of such system makes it hard to design them without defects. [47] puts it as follows:

> For programmers in industrial software development, the residual software defect ratio (the number of latent faults that remain in the code at the end of the development process) is normally somewhere between 0.5 and 5 defects per one thousand lines of non-comment source code [. . .]
>
> So without knowing anything about the particulars of a given industrially produced software product, one thing is generally safe to assume: it has bugs. The same is of course true for all industrial products, but what makes the problem unique in software is that the effects of even very minor programming mistakes can cause major system failures. It is very hard to contain the potential effects of a software defect, especially in distributed systems software [. . .]
>
> Even at a low residual defect density of 0.1 defect per one thousand lines of code, a ten million line source package will have an expected 10^3 latent defects. To reduce this number, we need to devise complementary analysis and testing techniques. It should also be noted that no-one really knows how many latent defects there *really* are in any given software product. All we can tell is how many of these defects eventually lead to customer complaints, in the years following product delivery. The industry average of 0.5 to 5 defects per one thousand lines of code is based on a typical count of the typical numbers of those customer complaints. We can suspect that the *true* number of latent defects is at least an order of magnitude higher, more likely in the range of 0.5 to 5 defects per one hundred lines of

> source code. Looking for latent software defects, then, is not quite like looking for needles in a haystack. Almost any new technique that differs sufficiently from traditional testing should be expected to intercept enough extra defects to justify its application.

Especially difficult is the development of concurrent systems, because they are generally harder to understand, and errors in them often do not show up during the testing. For example, even if due to a bug several processes can simultaneously access a critical section causing a race condition, this might happen quite rarely when the length of this critical section is short or this part of the code is executed relatively infrequently. Thus, such mistakes can easily go through the testing phase undetected and show up only after a long period of exploitation. The situation is in fact even more complicated, since concurrency related bugs are often *unrepeatable*: even if such a bug is detected during a particular testing run, it can well not occur during the next runs, making it difficult to locate it.

Therefore, conventional testing is not sufficient for establishing the correctness of such systems, and some kind of *formal verification* is required, e.g., in the form of a mathematical proof of correctness. (An alternative approach would be to use a methodology guaranteeing that the created system is 'correct by design' which does not require any validation at all, but in practice such methodologies are, in most cases, either impossible due to algorithmic undecidability or hard to implement.)

Traditionally, the correctness of a program is understood as termination with the correct result. But many safety-critical control systems are in fact *reactive systems*, i.e., non-terminating systems which maintain an ongoing interaction with their environment. Moreover, unlike traditional sequential programs, such systems are often composed of a set of concurrent processes and thus are *non-deterministic*: different execution *scenarios* (sequences of performed actions) are possible, depending on, e.g., how the operational system schedules those processes.

Several formal models of concurrent computation have been proposed so far. The most widely used are *process algebras* and *Petri nets*. The former are directly related to actual programming languages and are *compositional* by definition, i.e., larger systems can be composed from smaller ones in a structural way. They come with a variety of algebraic laws, which can be used to manipulate system specifications. The advantages of the latter formalism are simplicity and more intuitive semantics of execution: global states and global activities are not basic notions, but are derived from their local counterparts. Petri nets are one of relatively few formalisms admitting the *true concurrency semantics*, i.e., they can model concurrent execution of several actions directly, in contrast to the *interleaving semantics* of concurrency, where such an execution is modelled by a set of sequential runs, each of which is a permutation of these actions.[1] Moreover,

[1] Though some process algebras also admit the true concurrency semantics, it is usually given in a form resembling Petri net unfoldings (see, e.g., [71]).

Petri nets come with an intuitive graphical representation, and have useful links to graph theory and linear algebra. Some work has been done to combine the advantages of both formalisms (see, e.g., [3–5, 33, 34, 67]).

Classical logic is not well-suited for specifying and proving properties of concurrent computations, and [81] proposed to use *temporal logic* as a formalism suitable for this purpose. But even in such logics a formal proof of correctness can be much longer than the original system specification, and thus a hand-made proof itself is subject to errors. The only viable option is computer-aided verification.

The main two methods are *theorem proving* (see, e.g., [38]), in which a computer is used to derive a logic proof of correctness, and *model checking* (see, e.g., [13]), in which the verification of a system is carried out using a finite representation of its state space (sometimes a combination of these two methods is used). The former, while being very general, is semi-automatic and requires considerable human interaction, which is the main obstacle to the widespread use of this method. This problem is inherent for this method due to the classic incompleteness results for first-order logics (see, e.g., [25, 39, 83, 90]) and the algorithmic complexity of exploring the space of possible proofs. In this thesis we concentrate on the latter method, which often is completely automatic.

The main drawback of model checking is that it suffers from the *state space explosion* problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To alleviate this problem, a number of techniques have been proposed. They can roughly be classified as aiming at an implicit compact representation (e.g., in the form of a *binary decision diagram*, or *BDD*, see [8]) of the full state space of a reactive concurrent system, or at an explicit generation of its reduced (though sufficient for a given verification task) representation (e.g., *abstraction* ([12]) and *partial order reduction* ([37]) techniques). Among them, a prominent technique is McMillan's (finite prefixes of) Petri net unfoldings (see, e.g., [30, 31, 74, 85]). It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net. More precisely, given a Petri net Σ, the unfolding technique aims at building a labelled acyclic net Unf_Σ (a *prefix*) satisfying two key properties:

- *Completeness.* Each reachable marking of Σ is represented by at least one 'witness', i.e., one marking of Unf_Σ reachable from its initial marking. Similarly, for each possible firing of a transition at any reachable state of Σ there is a suitable 'witness' event in Unf_Σ.

- *Finiteness.* The prefix is finite and thus can be used as an input to model checking algorithms, e.g., those searching for deadlocks.

A prefix satisfying these two properties can be used for model checking as a condensed representation of the state space of a system. Indeed, it turns out that often such prefixes are exponentially smaller than the corresponding reachability graphs, especially if the system at hand exhibits a lot of concurrency.

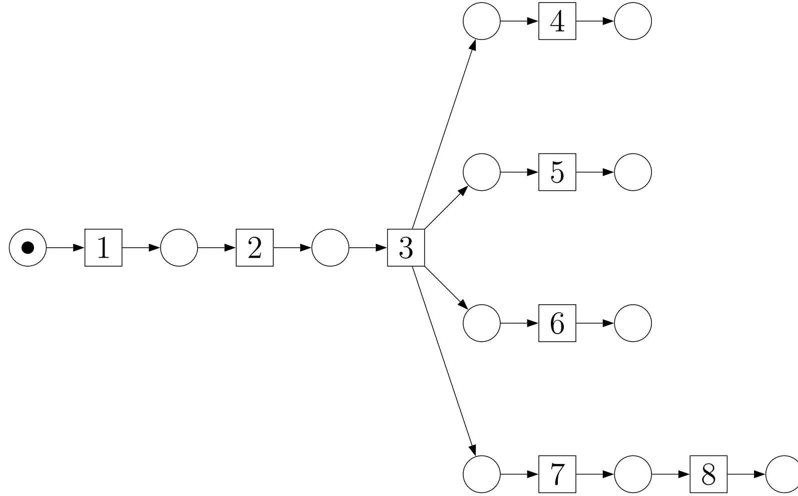

Figure 1: A Petri net describing the dependencies between the chapters.

Traditionally, prefix-based model checking is typically done in two steps: (i) generating a finite and complete prefix, and (ii) checking a relevant property on the generated prefix.[2] In this thesis, we address both these issues.

Organization of the Thesis

The thesis is organized as follows.

Chapter 1 provides the basic notions concerning Petri nets and their branching processes.

Chapter 2 discusses the theory of canonical prefixes of Petri net unfoldings.

Chapter 3 describes the test bench and the set of benchmarks used in further chapters to evaluate the algorithms developed there.

Chapter 4 addresses the issue of efficient computation of possible extensions of a prefix.

Chapter 5 presents a parallel unfolding algorithm.

Chapter 6 generalizes the prefix-based model checking technique to a class of high-level Petri nets.

[2] Another possible approach would be to check the property while generating a prefix, but this approach is largely the topic of future research, and is not discussed in the thesis.

Chapter 7 describes how integer programming can be employed for efficiently model checking a Petri net, using a finite and complete prefix of its unfolding.

Chapter 8 shows how this technique can be applied to check state encoding properties of specifications of asynchronous circuits.

The Petri net shown in Figure 1 describes the interdependencies between the chapters. Any of its runs gives a sequence in which the chapters could be read. Its unfolding (coinciding with the net itself) yields a partial order on the chapters. For example, Chapter 1 should be read before Chapter 2, whereas Chapters 4 and 5 can be read in any order (one can even attempt to read them concurrently). The reachability graph of this Petri net is shown in Figure 2. Its directed paths starting from the initial state still give valid sequences of chapters, but it is rather hard to comprehend because of its size. At least, it is not immediately obvious from it which chapters are dependent and which are not. Thus the state space explosion problem and the advantages of the unfolding technique can be clearly seen even on this simple example!

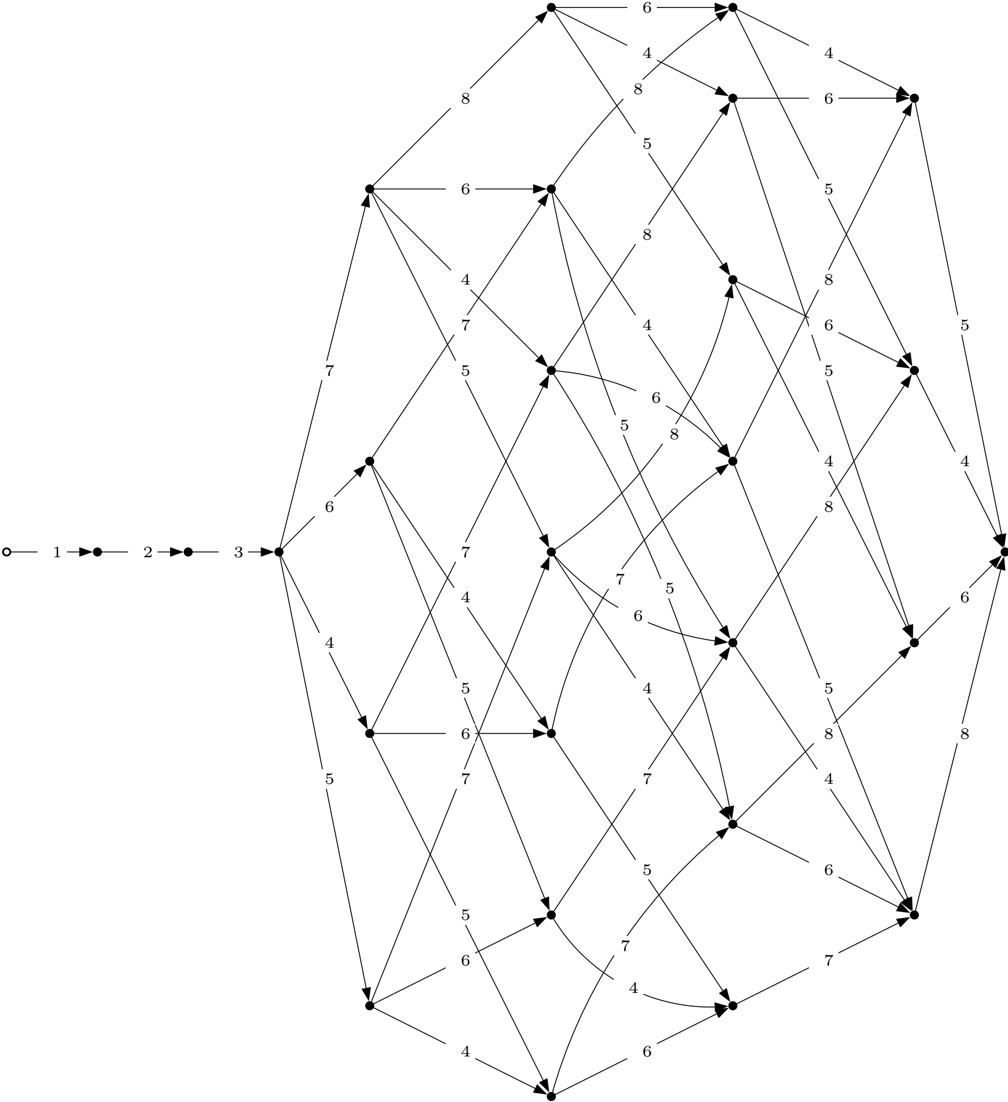

Figure 2: The reachability graph of the Petri net shown in Figure 1.

Chapter 1

Basic Notions

In this chapter, we present the basic notions which will be used throughout the thesis.

1.1 Multisets

A *multiset* over a set X is a function $\mu : X \to \mathbb{N} \stackrel{\text{df}}{=} \{0, 1, 2, \ldots\}$. Note that any subset of X may be viewed (through its characteristic function) as a multiset over X. We denote $x \in \mu$ if $\mu(x) \geq 1$, and for two multisets over X, μ and μ', we write $\mu \leq \mu'$ if $\mu(x) \leq \mu'(x)$ for all $x \in X$. We will use $\emptyset$ to denote the *empty multiset* defined as $\emptyset(x) \stackrel{\text{df}}{=} 0$, for all $x \in X$. Moreover, a finite multiset may be represented by explicitly listing its elements (perhaps, with coefficients separated by $*$) between the $\{\!|\ldots|\!\}$ brackets. For example $\{\!|2*a, b, 5*c|\!\}$ denotes the multiset μ such that $\mu(a) = 2$, $\mu(b) = 1$, $\mu(c) = 5$ and $\mu(x) = 0$, for $x \in X \setminus \{a, b, c\}$.

The *sum of two multisets* μ' and μ'' over X is given by $(\mu' + \mu'')(x) \stackrel{\text{df}}{=} \mu'(x) + \mu''(x)$, the *difference* by $(\mu' - \mu'')(x) \stackrel{\text{df}}{=} \max\{0, \mu'(x) - \mu''(x)\}$, and the *intersection* by $(\mu' \cap \mu'')(x) \stackrel{\text{df}}{=} \min\{\mu'(x), \mu''(x)\}$, for all $x \in X$. A multiset μ is *finite* if there are finitely many $x \in X$ such that $\mu(x) \geq 1$. In such a case, the *cardinality* of μ is defined as $|\mu| \stackrel{\text{df}}{=} \sum_{x \in X} \mu(x)$.

The notation $\{\!|h(x) \mid x \in \mu|\!\}$, or, alternatively, $h\{\!|\mu|\!\}$, where μ is a multiset over X and $h : X \to Y$ is a function, will be used to denote the multiset μ' over Y such that

$$\mu'(y) \stackrel{\text{df}}{=} \sum_{x \in X \wedge h(x) = y} \mu(x) \ .$$

For example, $\{\!|x^2{+}1 \mid x \in \{\!|-1, 2*0, 1|\!\}|\!\} = h\{\!|-1, 2*0, 1|\!\} = \{\!|2*1, 2*2|\!\}$, where $h(x) \stackrel{\text{df}}{=} x^2 + 1$.

If $f : X \to \mathbb{Z} \stackrel{\text{df}}{=} \{0, \pm 1, \pm 2, \ldots\}$ is a function and μ is a multiset over X then

$$\sum_{x \in \mu} f(x) \stackrel{\text{df}}{=} \sum_{x \in X} \mu(x) f(x)$$

if the latter sum is defined.

Let $\ll$ be some strict total order on set X. Then the lexicographical order $\ll_{lex}$ and the size-lexicographical order $\ll_{sl}$ on the multisets over X are defined as follows:

- $\mu' \ll_{lex} \mu''$ if there exists $x \in X$ such that $\mu'(x) < \mu''(x)$ and, for all $y \in X$ such that $y \ll x$, $\mu'(y) = \mu''(y)$.
- $\mu' \ll_{sl} \mu''$ if either $|\mu'| < |\mu''|$ or $|\mu'| = |\mu''|$ and $\mu' \ll_{lex} \mu''$.

1.2 Noetherian induction

In this section we discuss the principle of Noetherian induction (see [14, 15] for more details), which will be used in Chapter 2.

Let $\lhd$ be a strict partial order on a set X. The set of $\lhd$-predecessors of an element $x \in X$ is defined as $\{x' \in X \mid x' \lhd x\}$. A set $X' \subseteq X$ is called a $\lhd$-*chain* in X if the restriction of $\lhd$ to X' is a total order. A $\lhd$-chain $\{x_1, x_2, \ldots\} \subseteq X$ is *descending* if

$$\cdots \lhd x_2 \lhd x_1 \ .$$

The order $\lhd$ is *well-founded* if every descending chain in X has only finitely many elements.

Proposition 1.1 (Principle of Noetherian Induction). *Let $\lhd$ be a strict well-founded partial order on a set X, and X' be a subset of X which contains an element of X whenever it contains all its $\lhd$-predecessors. Then $X' = X$.*

This principle allows one to prove a property P for all elements of X by showing that it holds for each element of X whenever it holds for all its $\lhd$-predecessors. Indeed, it suffices to apply the above proposition taking $X' \stackrel{\text{df}}{=} \{x \in X \mid P(x)\}$. Similarly, one can define a predicate or function for each element $x \in X$ assuming that it has already been defined for all $\lhd$-predecessors of x. The principle of Noetherian induction ensures that it will be defined for all the elements of X.

1.3 Petri nets

A *net (with weighted arcs)* is a triple $N \stackrel{\text{df}}{=} (P, T, W)$ such that P and T are disjoint sets of *places* and *transitions* respectively, and W is a multiset over $(P \times T) \cup (T \times P)$ called the *weight function*. The net N is called *ordinary* if W is a set; in such a case, W can be considered as a *flow relation* on $(P \times T) \cup (T \times P)$. A *marking* of N is a multiset M over P (note that M is finite whenever P is), and the set of all markings of N will be denoted by $\mathcal{M}(N)$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the weight function by arcs with the indicated weight (we do not draw

arcs whose weight is 0, and we do not indicate the weight if it is 1), and markings are shown by placing tokens within circles. The multisets ${}^\bullet z \stackrel{\mathrm{df}}{=} \{\!| y \mid (y,z) \in W |\!\}$ and $z^\bullet \stackrel{\mathrm{df}}{=} \{\!| y \mid (z,y) \in W |\!\}$, denote the *pre-* and *postset* of $z \in P \cup T$. (Note that if N is an ordinary net then both ${}^\bullet z$ and $z^\bullet$ are sets.) For ordinary nets we also define ${}^\bullet Z \stackrel{\mathrm{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\mathrm{df}}{=} \bigcup_{z \in Z} z^\bullet$, where Z is a set of nodes. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma \stackrel{\mathrm{df}}{=} (N, M_0)$ comprising a finite net $N = (P, T, W)$ and an *initial marking* $M_0 \in \mathcal{M}(N)$. A transition $t \in T$ is *enabled* at a marking M if ${}^\bullet t \leq M$. Such a transition can be *fired*, leading to the marking $M' \stackrel{\mathrm{df}}{=} M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$. The set of *reachable markings* of Σ is the smallest (w.r.t. $\subseteq$) set $\mathcal{RM}(\Sigma)$ containing M_0 and such that if $M \in \mathcal{RM}(\Sigma)$ and $M[t\rangle M'$, for some $t \in T$ and $M' \in \mathcal{M}(N)$, then $M' \in \mathcal{RM}(\Sigma)$. For a finite sequence of transitions $\sigma = t_1 \dots t_k$, we write $M[\sigma\rangle M'$ if there are markings $M_1, \dots, M_{k+1}$ such that $M_1 = M$, $M_{k+1} = M'$, and $M_i[t_i\rangle M_{i+1}$, for all $i = 1, \dots, k$.

A marking is *deadlocked* if it does not enable any transitions. Σ is *deadlock-free* if none of its reachable markings is deadlocked. A transition t is *dead* if no reachable marking enables it. M' *covers* M, if $M \leq M'$. A marking M is *coverable* if there exists $M' \in \mathcal{RM}(\Sigma)$ such that M' covers M.

Σ is k*-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. Moreover, Σ is *bounded* if it is k-bounded for some $k \in \mathbb{N}$. One can show that the set $\mathcal{RM}(\Sigma)$ is finite iff Σ is bounded.

Note that if a safe net system Σ has arcs of weight more than 1, the transitions incident to them can never become enabled, and so can be removed (together with their incoming and outgoing arcs) without changing the behaviour of Σ. Thus, one can assume that the underlying nets of safe net systems are ordinary.

Places $p_1, \dots, p_k$ of a net system Σ are *mutually exclusive* if no reachable marking puts tokens in more than one of them, i.e., for every $M \in \mathcal{RM}(\Sigma)$, $M(p_i) \geq 1$ implies $M(p_j) = 0$, for all $j \in \{1, \dots, k\} \setminus \{i\}$.

As an example, consider the net system Σ_{p-c} shown in Figure 1.1. At the initial marking $M_0 = \{\!| p_1, 2 * p_2, p_3 |\!\}$, the only enabled transition is t_1. It can fire leading to the marking $M_1 = \{\!| 2 * p_2, p_3, p_4 |\!\}$, i.e., $M_0[t_1\rangle M_1$. At M_1, the only enabled transition is t_2, and $M_1[t_2\rangle M_2 = \{\!| p_1, p_2, p_3, p_5 |\!\}$. M_2 enables two transitions, t_1 and t_3, any of which can fire, and so on. One can show (e.g., by computing the set $\mathcal{RM}(\Sigma_{p-c})$ of all reachable markings, which is finite in this case) the following:

- Σ_{p-c} is deadlock-free, i.e., none of its reachable marking is deadlocked.
- Σ_{p-c} is 2-bounded but not safe.
- The places p_1 and p_4 of Σ_{p-c} are mutually exclusive, whereas the places p_2 and p_5 are not (note that, e.g., M_2 marks both p_2 and p_5).

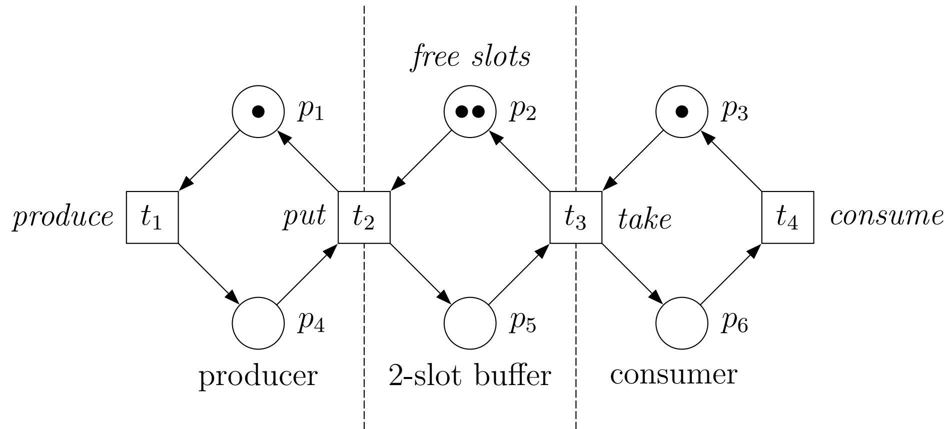

Figure 1.1: A Petri net model of a simple producer-consumer system.

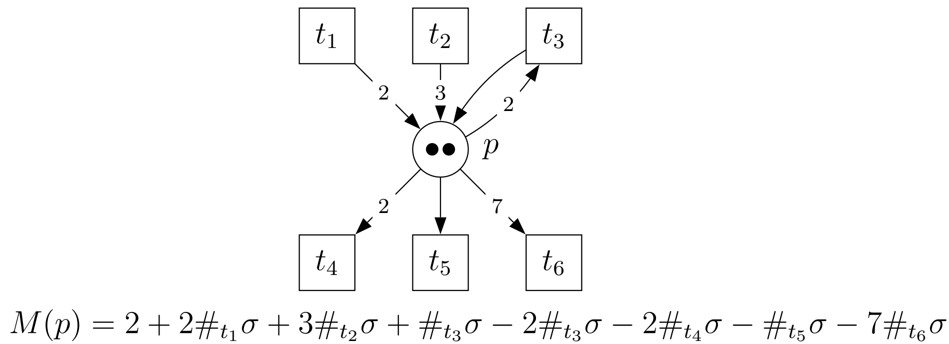

Figure 1.2: Marking equation (only one place with its environment and initial marking is shown).

1.4 Marking equation

Let $\Sigma = (N, M_0)$ be a net system, p be one of its places, and σ be a finite execution sequence of Σ such that $M_0[\sigma\rangle M$. By counting the tokens brought to and taken from p by the transitions in σ it is possible to calculate $M(p)$ as follows:

$$M(p) = M_0(p) + \sum_{t \in T} W((t, p))\#_t\sigma - \sum_{t \in T} W((p, t))\#_t\sigma \ ,$$

where $\#_t\sigma$ denotes the number of times a transition t occurs in σ (see Figure 1.2). This is a linear equation which holds for every place of Σ. It can be written in matrix form as follows.

Let $p_1, \ldots, p_m$ and $t_1, \ldots, t_n$ be respectively the places and transitions of Σ. One can identify a marking M of Σ with the vector $(M(p_1), \ldots, M(p_m))$. The *incidence matrix* of Σ is an $m \times n$ matrix $\mathfrak{I}_\Sigma = (\mathfrak{I}_{\Sigma ij})$ such that, for all $i \leq m$

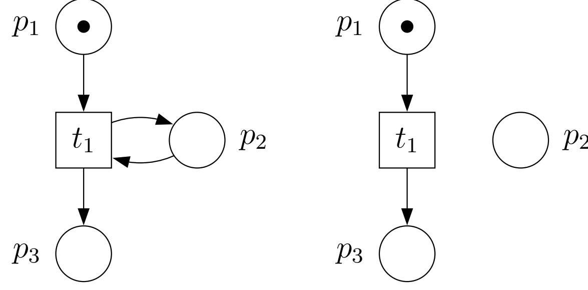

Figure 1.3: Two net systems which have distinct sets of reachable markings but are indistinguishable by the marking equation. Note that these net systems have the same incidence matrix and the same initial marking, and so the same set of solutions of the marking equation.

and $j \le n$,

$$\mathfrak{I}_{\Sigma ij} \stackrel{\text{df}}{=} W((t_j, p_i)) - W((p_i, t_j)) \ .$$

The *Parikh vector* of σ is a vector $x_\sigma = (\#_{t_1}\sigma, \ldots, \#_{t_n}\sigma)$. One can show that if σ is an execution sequence such that $M_0[\sigma\rangle M$ then $M = M_0 + \mathfrak{I}_\Sigma \cdot x_\sigma$.

This provides a motivation for investigating the feasibility (or solvability) of the following *marking equation* (see also [76–78, 88]):

$$\begin{cases} M = M_0 + \mathfrak{I}_\Sigma \cdot x \\ M \in \mathbb{N}^m \text{ and } x \in \mathbb{N}^n \ . \end{cases} \tag{1.1}$$

If M is fixed then the feasibility of the above system is a necessary (but, in general, not sufficient) condition for M to be reachable from M_0. This is so because the Parikh vector of every execution sequence σ such that $M_0[\sigma\rangle M$ is a solution of (1.1), but, in general, (1.1) can have solutions which do not correspond to any execution sequence of Σ (see Figure 1.3). Therefore, the set of markings M for which (1.1) is feasible is an overapproximation of the set of reachable markings of Σ.

A vector $x \in \mathbb{N}^n$ is Σ-*compatible* if it is the Parikh vector of some execution sequence of Σ. Each Σ-compatible vector is a solution of the marking equation for some reachable marking M, but, in general, not every solution of (1.1) is Σ-compatible. However, for the class of acyclic nets (in particular, *branching processes* defined below), this equation provides an *exact* characterization of the set of reachable markings (see, e.g., [77, 78]) — the fact which is crucial for the approach proposed in Chapter 7.

1.5 Branching Processes

In this section, we recall notions related to net unfoldings (see also [26, 30, 31, 60, 61, 85]).

Two nodes (places or transitions), y and y', of an ordinary net $N = (P, T, W)$ are in *conflict*, denoted by $y\#y'$, if there are distinct transitions $t, t' \in T$ such

that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation W, denoted by $\preceq$. A node y is in *self-conflict* if $y\#y$.

An *occurrence net* is an ordinary net $ON \stackrel{\text{df}}{=} (B, E, G)$, where B is a set of *conditions* (places), E is a set of *events* (transitions) and G is a flow relation, satisfying the following: ON is acyclic (i.e., $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y\#y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the transitive closure of G. $Min(ON)$ will denote the set of minimal (w.r.t. $\prec$) elements of $B \cup E$. The relation $\prec$ is the *causality relation*. A *$\prec$-chain* of events is a finite or infinite sequence of events such that for each two consecutive events, e and f, it is the case that $e \prec f$. Two nodes are *concurrent*, denoted $y \; co \; y'$, if neither of $y\#y'$, $y \preceq y'$, $y' \preceq y$ holds. Two events e and f are *separated* if there is an event g such that $e \prec g \prec f$.

Definition 1.2 (Branching process). A *homomorphism* from an occurrence net $ON = (B, E, G)$ to a net system Σ is a mapping $h : B \cup E \to P \cup T$ such that

- $h(B) \subseteq P$ and $h(E) \subseteq T$ (conditions are mapped to places, and events to transitions).
- For each $e \in E$, $h\{\!|{}^\bullet e|\!\} = {}^\bullet h(e)$ and $h\{\!|e^\bullet|\!\} = h(e)^\bullet$ (transition environments are preserved).
- $h\{\!|Min(ON)|\!\} = M_0$ (minimal conditions correspond to the initial marking).
- For all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$ (there is no redundancy).

A *branching process* of Σ is a pair $\pi \stackrel{\text{df}}{=} (ON, h)$ such that ON is an occurrence net and h is a homomorphism from ON to Σ. $\diamond$

If an event e is such that $h(e) = t$ then we will often refer to it as being *t-labelled*. A branching process $\pi' = (ON', h')$ of Σ is a *prefix of a branching process* $\pi = (ON, h)$, denoted $\pi' \sqsubseteq \pi$, if $ON' = (B', E', G')$ is a subnet of $ON = (B, E, G)$ containing all minimal elements and such that: (i) if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; (ii) if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and (iii) h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process Unf_Σ^{max}, called the *unfolding* of Σ (see [26]).

Sometimes it is convenient to start a branching process with a (virtual) initial event $\perp$, which has the postset $Min(ON)$, empty preset, and no label; we will henceforth assume that such an event exists, without drawing it in figures or treating it explicitly in algorithms. The *depth of an event* e is defined as the length of the longest $\prec$-chain of events ending at e.

An example of a safe net system and two of its branching prefixes is shown in Figure 1.4, where the homomorphism h is indicated by the labels of the nodes. The process in Figure 1.4(b) is a prefix of that in Figure 1.4(c).

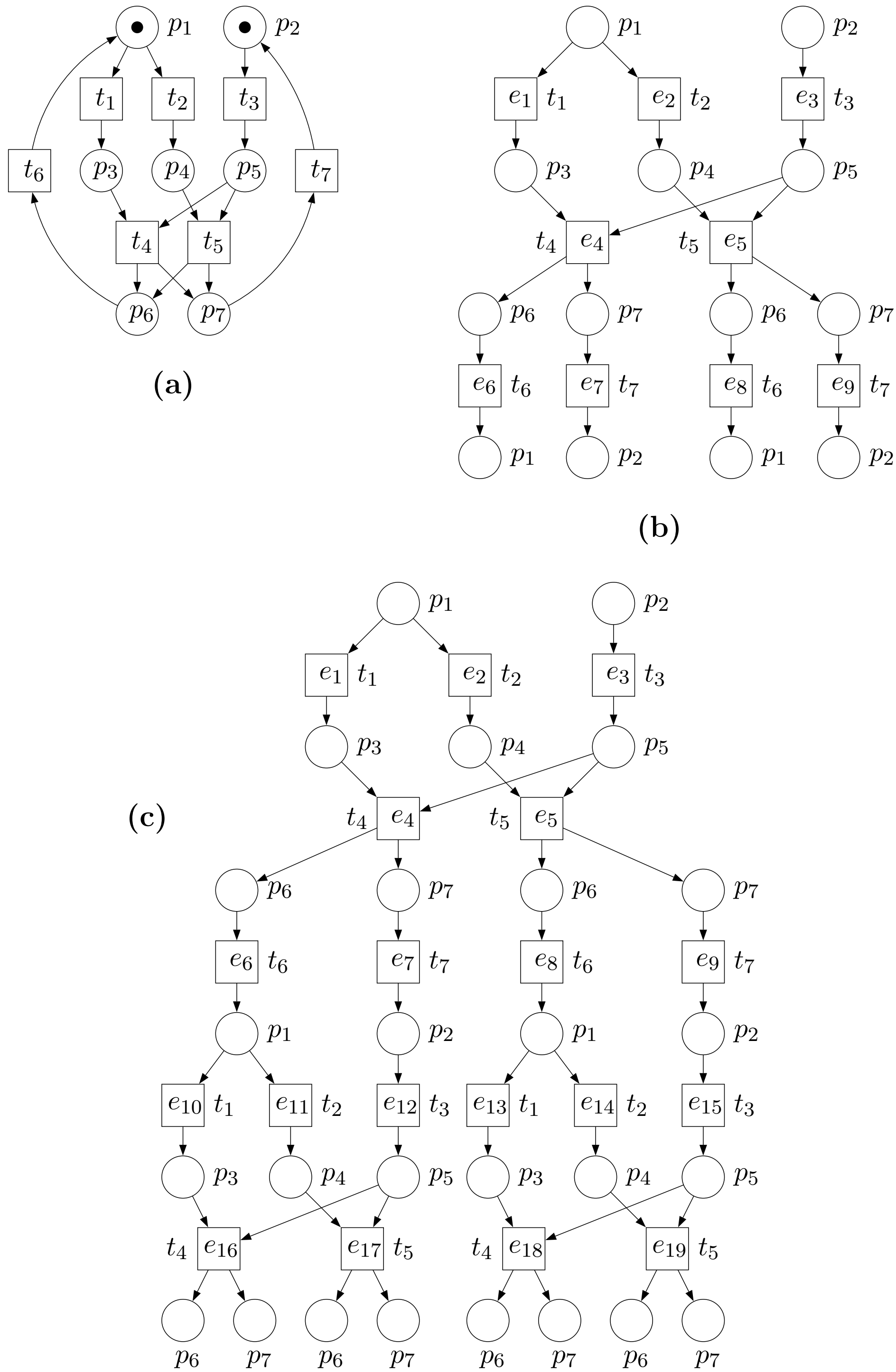

Figure 1.4: A net system **(a)** and two of its branching processes **(b,c)**.

Configurations and cuts

A *configuration* of an occurrence net ON is a set of events C such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$; since we assume the initial event $\bot$, we additionally require that $\bot \in C$. For every event $e \in E$, the configuration $[e] \stackrel{\mathrm{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of e, and $\langle e \rangle \stackrel{\mathrm{df}}{=} [e] \setminus \{e\}$ denotes the set of *causal predecessors* of e. Moreover, for a set of events E' we denote by $C \oplus E'$ the fact that $C \cup E'$ is a configuration and $C \cap E' = \emptyset$. Such an E' is a *suffix* of C, and $C \oplus E'$ is an *extension* of C.

The set of all finite (resp. local) configurations of a branching process π will be denoted by $\mathcal{C}^{\pi}_{fin}$ (resp. $\mathcal{C}^{\pi}_{loc}$), and we will drop the superscript π in the case $\pi = Unf^{max}_{\Sigma}$.

A set of events E' is *downward-closed* if all causal predecessors of the events in E' also belong to E'. Such a set *induces* a unique branching process π whose events are exactly the events in E', and whose conditions are the conditions incident to the events in E' (including $\bot$).

A set of conditions B' such that for all distinct $b, b' \in B'$, b *co* b', is called a *co-set*. A *cut* is a maximal (w.r.t. $\subseteq$) co-set. Every marking reachable from $Min(ON)$ is a cut.

Let C be a finite configuration of a branching process π. Then the set

$$Cut(C) \stackrel{\mathrm{df}}{=} \Big(\bigcup_{e \in C} e^{\bullet}\Big) \setminus \Big(\bigcup_{e \in C} {}^{\bullet}e\Big)$$

is a cut (note that $\bot \in C$); moreover, the multiset of places $Mark(C) \stackrel{\mathrm{df}}{=} h\{\!|Cut(C)|\!\}$ is a reachable marking of Σ, called the *final marking* of C. A marking M of Σ is *represented* in π if there is $C \in \mathcal{C}^{\pi}_{fin}$ such that $M = Mark(C)$. Every marking represented in π is reachable in the original net system Σ, and every reachable marking of Σ is represented in Unf^{max}_{Σ}.

The following definition is crucial for descriptions of unfolding algorithms.

Definition 1.3. Let π be a branching process of Σ and e be an event of π. A *possible extension* of π is a pair (t, D), where D is a co-set in π and t is a transition of Σ, such that $h(D) = {}^{\bullet}t$ and π contains no t-labelled event with preset D. A possible extension (t, D) of π is a (π, e)*-extension* if $D \cap e^{\bullet} \neq \emptyset$, and e and (t, D) are not separated. $\diamond$

The pair (t, D) is an event used by unfolding algorithms to extend π, and we will take it as being t-labelled and having D as its preset. PotExt(π) and UpdatePotExt(π, e) will denote the set of possible extensions of π and the set of (π, e)-extensions, respectively.

Unfolding algorithms, starting from a branching process induced by the initial event, repeatedly apply the function PotExt to extend it, until certain stopping criterion is met. An efficient method to compute the value of PotExt is to apply the function UpdatePotExt to each event generated on the previous step.

If S is a set of possible extensions of a prefix π, we will denote by $\pi \oplus S$ a prefix obtained by adding the events from S together with their postsets to π. Note that by the definition of a possible extension, a branching process cannot contain a t-labelled event with preset D, for any $(t, D) \in S$.

As an example, consider the branching process $\tilde{\pi}$ induced by events e_1–e_5 in Figure 1.4(c). Then $\textsc{PotExt}(\tilde{\pi}) = \{e_6, e_7, e_8, e_9\}$, $\textsc{UpdatePotExt}(\tilde{\pi}, e_4) = \{e_6, e_7\}$, and $\tilde{\pi} \oplus \{e_6, e_8\}$ is the branching process induced by events e_1–e_6 and e_8.

Chapter 2

Canonical Prefixes

Though Petri net unfoldings are infinite whenever the original net systems have infinite runs, it turns out that they can often be truncated in such a way that the resulting prefixes, though finite, contain enough information to decide a certain behavioural property, e.g., deadlock freeness. We then say that the prefixes are *complete* for this property. Historically (see, e.g., [29–31, 74, 75, 85]), the completeness was intuitively understood as an ability to generate the full unfolding from the given prefix, i.e., every reachable marking and every transition firing of the original net system must be represented in the prefix. We examine and generalize this notion of completeness, making it more applicable to practical model checking. There are two fundamental issues which we wish to address in this chapter, namely the precise *semantical* meaning of completeness, and the *algorithmic* problem of generating complete prefixes.

This chapter is based on the results developed in [60, 61].

2.1 Semantical meaning of completeness

A direct motivation to re-examine the issue of completeness was provided by an experience of dealing with unfoldings of Signal Transition Graphs (STGs) in [62, 63], used to specify the behaviour of asynchronous circuits. Briefly, an STG (see [86]) is a Petri net together with a set of binary signals (variables), which can be set or reset by transition firings. A transition can either change the value of one specific signal, or affect no signals at all. Thus, the current *encoding* (the vector of signal values) depends not on the current marking, but rather on the sequence of transition firings that leads to it. In effect, one is interested in a 'combined' system state which includes both the current marking and the current values of all binary signals. Therefore, in order to ensure that a prefix represents the entire state space, some additional information (in this case, the valuation of signal variables) must be taken into account. Clearly, the completeness as sketched above does not guarantee this (see the example in Figure 8.3).

We soon found out that the situation can also be a totally opposite one, i.e., the standard notion of completeness can be unnecessarily strong. As an

example, one can consider the building of a prefix when there is a suitable notion of symmetric (equivalent) markings, as described in [24]. The idea is then to ensure that each marking of Σ is represented in Unf_Σ either directly or by a symmetric marking. Such an approach may significantly reduce the size of the prefix.

Having analyzed examples like these, we have concluded that the original notion of completeness, though sufficient for certain applications, may be too crude and inflexible if one wants to take into consideration more complicated semantics of concurrent systems, or their inherent structural properties.

2.2 Algorithmics of prefix generation

The essential feature of the existing unfolding algorithms (see, e.g., [29–31,45,46, 74,85]) is the use of cut-off events, beyond which the unfolding starts to repeat itself and so can be truncated without loss of information. So far, cut-off events were considered as an algorithm-specific issue, and were defined w.r.t. the part of the prefix already built by an unfolding algorithm (in other words, at run-time). Such a treatment was quite pragmatic and worked reasonably well. But, in more complicated situations, the dynamic notion of a cut-off event may hinder defining appropriate algorithms and, in particular, proving their correctness. This has become apparent when dealing with a parallel algorithm for generating prefixes in [45,46], where the degree of possible non-determinism brought up both these issues very clearly. As a result, the correctness proof given there is long and hard to comprehend. To conclude, the algorithm-dependent notion of a cut-off event is becoming increasingly difficult to manage.

There is also an important aspect linking cut-off events and completeness, which was somewhat overlooked in previous works. To start with, the notion of a complete prefix given in [30,31] did not mention cut-off events at all. But, with the development of model checking algorithms based on unfoldings, it appeared that cut-off events are heavily employed by almost all of them. Indeed, the deadlock detection algorithm presented in [74] is based on the fact that a Petri net is deadlock-free iff each configuration of its finite and complete prefix can be extended to one containing a cut-off event, i.e., a Petri net has a deadlock iff there is a configuration which is in conflict with all cut-off events. The algorithms presented in [40–42,44,52–55,77] use the fact that there is a certain correspondence between the deadlocked markings of the original net and the deadlocked markings of a finite and complete prefix, and cut-off events are needed to distinguish the 'real' deadlocks from the 'fake' ones, introduced by truncating the unfolding. Moreover, those algorithms need a stronger notion of completeness than the one presented in [30,31], in order to guarantee that deadlocks in the prefix do correspond to deadlocks in the original Petri net. Indeed, according to the notion of completeness presented in [30,31], a marking M enabling a transition t may be represented by a deadlocked configuration C in a complete prefix, as

long as there is another configuration C' representing this marking and enabling an instance of t. This means that the prefix may contain a deadlock, which does not correspond to any deadlock in the original net system (see Figure 2.1). Since all these algorithms make certain assumptions about the properties of a prefix with cut-off events, it is natural to formally link cut-off events with the notion of completeness, closing up a rather uncomfortable gap between theory and practice.

2.3 The new approach

In order to address the issues of semantical meaning of completeness and algorithmic pragmatics relating to Petri net unfolding prefixes, we propose a parametric setup in which questions concerning, e.g., completeness and cut-off events, could be discussed in a uniform and general way. One parameter captures the information we intend to retain in a complete prefix, while the other two specify under which circumstances a given event can be designated as a cut-off event. Crucially, we decided to shift the emphasis from markings to the execution histories of Σ, and the former parameter, a suitably defined equivalence relation $\approx$, specifies which executions can be regarded as equivalent. Intuitively, one has to retain at least one representative execution from each equivalence class of $\approx$. (The standard case in [29–31,74,85] is then covered by regarding two executions as equivalent iff they reach the same marking.)

For efficiency reasons, the existing unfolding algorithms usually consider only local configurations when deciding whether an event should be designated as a cut-off event. But one can also consider arbitrary finite configurations for such a purpose if the size of the resulting prefix is of paramount importance (see, e.g., [43]). As a result, the final definition of the setup, called here a *cutting context*, contains a parameter which specifies for each event precisely those configurations which can be used to designate it as a cut-off event. For a given equivalence relation $\approx$, we then define what it means for a prefix to be complete. In essence, we require that all equivalence classes of $\approx$ are represented, and that any history involving no cut-off events can be extended (by a single step) in exactly the same way as in the full unfolding.

The definition of a cutting context leads to the central result of this chapter, namely the *algorithm-independent* notion of a cut-off event and the related unique *canonical* prefix; the latter is shown to be complete w.r.t. the new notion of completeness. Though the canonical prefix is always complete, it may still be infinite, making it unusable for model checking. We therefore investigate what guarantees the finiteness of the canonical prefix and, in doing so, formulate and prove a version of König's Lemma for branching processes of (possibly unbounded) Petri nets.

To summarize, this chapter addresses both semantical and algorithmic problems using a single device, namely the canonical prefix. The theoretical notion

of a complete prefix is useful as long as it can be the basis of a practical prefix-building algorithm. We show that this is indeed the case, generalizing the already proposed unfolding algorithm presented in [30, 31] as well as the parallel algorithm from [45, 46]. We believe that the approach presented in this chapter results in a more elegant framework for investigating issues relating to unfolding prefixes, and provides a powerful and flexible tool to deal with different variants of the unfolding technique.

2.4 König's Lemma for branching processes

König's Lemma (see [66]) states that a finitely branching, rooted, directed acyclic graph with infinitely many nodes reachable from the root has an infinite path.[1] We prove here what can be regarded as a version of such a result for branching processes of Petri nets.

Proposition 2.1. *A branching process π of a net system Σ is infinite iff it contains an infinite $\prec$-chain of events.*

Proof. The 'if' part of the statement is trivial. To prove the 'only if' part, we observe that if π is infinite then, by the fact that branching on each event (including $\bot$) is finite, π comprises infinitely many events.

For every event e of π, let $d(e)$ be the depth of e, and E_k be the set of events of π with the depth not greater than k. By induction, one can show that, for every k, E_k is finite. Indeed, the base case is trivial, as $E_1 = \{\bot\}$. Now, assuming that E_k is finite we prove that E_{k+1} is finite. By the induction hypothesis and the finiteness of the set $e^\bullet$ for every event e (including $\bot$), $\bigcup_{e\in E_k} e^\bullet$ is finite. Since $\bigcup_{e\in E_{k+1}} {}^\bullet e \subseteq \bigcup_{e\in E_k} e^\bullet$ (note that $\bot \in E_k$ for every k) and ${}^\bullet e$ is non-empty for every event $e \neq \bot$, we get by the finiteness of Σ and the last property (irredundancy) in the Definition 1.2 of a branching process that E_{k+1} is finite. (Compare the proof of Proposition 4.8 in [31].)

Now, consider a graph $\mathcal{G}$ on the set of events of π, such that there is an arc from e to e' iff $e \prec e'$ and $d(e') = d(e) + 1$. Clearly, $\mathcal{G}$ is an infinite, rooted, directed acyclic graph with all its nodes being reachable from the root $\bot$. By the finiteness of E_k for every k, $\mathcal{G}$ is finitely branching, and so, by König's Lemma, there is an infinite path in $\mathcal{G}$. Clearly, such a path determines an infinite $\prec$-chain of events in π. □

It is worth noting that the above result is not a trivial corollary from the original König's Lemma, since conditions of a branching process can have infinitely many outgoing arcs.

[1] The graph may be such that, for every $n \in \mathbb{N}$, there is a node with more than n branches, but no node can have an infinite number of branches. Note also that the existence of an infinite path is a stronger property than the existence of arbitrary long finite paths.

2.5 Complete prefixes of Petri net unfoldings

As explained in the beginning of the chapter, there exist different methods of truncating Petri net unfoldings. The differences are related to the kind of information about the original unfolding one wants to preserve in the prefix, as well as to the choice between using either only local configurations (which can improve the running time of an algorithm), or all finite configurations (which can result in a smaller prefix). In addition, we need a more general notion of completeness for branching processes. In this section, we generalize the entire setup so that it will be applicable to different methods of truncating unfoldings and, at the same time, allow one to express the completeness w.r.t. properties other than marking reachability.

2.5.1 Cutting context

In order to cope with different variants of the technique for truncating unfoldings, we generalize the whole setup, using an abstract parametric model. The first parameter will determine the information we intend to preserve in a complete prefix (in the standard case, this is the set of reachable markings). The main idea behind it is to speak about finite configurations of Unf_{Σ}^{max} rather than reachable markings of Σ. Formally, the information to be preserved corresponds to the equivalence classes of some equivalence relation $\approx$ on $\mathcal{C}_{fin}$. The other parameters are more technical: they specify the circumstances under which an event can be designated as a cut-off event.

Definition 2.2. A *cutting context* is a triple

$$\Theta \overset{\text{df}}{=} \left(\approx\,,\, \lhd\,,\, \{\mathcal{C}_e\}_{e\in E}\right) \;,$$

where:

1. $\approx$ is an equivalence relation on $\mathcal{C}_{fin}$.
2. $\lhd$, called an *adequate* order (compare [31, Definition 4.5]), is a strict well-founded partial order on $\mathcal{C}_{fin}$ refining $\subset$, i.e., $C' \subset C''$ implies $C' \lhd C''$.
3. $\approx$ and $\lhd$ are *preserved by finite extensions*, i.e., for every pair of configurations $C' \approx C''$, and for every suffix E' of C', there exists[2] a finite suffix E'' of C'' such that
    (a) $C'' \oplus E'' \approx C' \oplus E'$, and
    (b) if $C'' \lhd C'$ then $C'' \oplus E'' \lhd C' \oplus E'$.

[2] Note that unlike [30, 31], we do not require that $E'' = I_1^2(E')$, where I_1^2 is the 'natural' isomorphism between the finite extensions of C' and C''. That isomorphism is defined only if $Mark(C') = Mark(C'')$, and thus cannot be used in the proposed generalized setup.

4. $\{\mathcal{C}_e\}_{e\in E}$ is a family of subsets of $\mathcal{C}_{fin}$. $\diamond$

The main idea behind the adequate order is to specify which configurations will be preserved in the complete prefix; it turns out that all $\lhd$-minimal configurations in each equivalence class of $\approx$ will be preserved. The last parameter is needed to specify the set of configurations used later to decide whether an event can be designated as a cut-off event. For example, $\mathcal{C}_e$ may contain all finite configurations of Unf_{Σ}^{max}, or, as it is usually the case in practice, only the local ones. We will say that a cutting context Θ is *dense* (*saturated*) if $\mathcal{C}_e \supseteq \mathcal{C}_{loc}$ (resp. $\mathcal{C}_e = \mathcal{C}_{fin}$), for all $e \in E$.

In practice, Θ is usually dense (or even saturated, see [43]), and at least the following cases of the adequate order $\lhd$ and the equivalence $\approx$ have been shown in the literature to be of interest:

Adequate orders

- $C' \lhd_m C''$ if $|C'| < |C''|$. This is the original McMillan's adequate order (see [30, 31, 75]). It is adequate for arbitrary net systems, but, in general, not total.
- $C' \lhd_{sl} C''$ if $h\{\!|C'|\!\} \ll_{sl} h\{\!|C''|\!\}$, where $\ll$ is an arbitrary total order on the transitions of the original net system. This is an adequate order for arbitrary net systems, proposed in [31]. Though it refines $\lhd_m$, it is, in general, not total.
- $C' \lhd_{erv} C''$ if either $C' \lhd_{sl} C''$ or $h\{\!|C'|\!\} = h\{\!|C''|\!\}$ and $C' \ll_f C''$, where $\ll$ is an arbitrary total order on the transitions of the original net system and $\ll_f$ is built upon $\ll$ as follows: $C' \ll_f C''$ if there exists $i \in \mathbb{N} \setminus \{0\}$ such that $h\{\!|F_i(C')|\!\} \ll_{sl} h\{\!|F_i(C'')|\!\}$ and, for every $j \in \{0, \ldots, i-1\}$, $h\{\!|F_j(C')|\!\} = h\{\!|F_j(C'')|\!\}$. Here $F_k(C)$ is the k-th level of the Foata normal form of C, defined as the set of events of C whose depth in the unfolding is k. It is shown in [30,31] that $\lhd_{erv}$ is a total adequate order for safe net systems. It refines both $\lhd_m$ and $\lhd_{sl}$.

Equivalence relations

- $C' \approx_{mar} C''$ if $Mark(C') = Mark(C'')$. This is the most widely used equivalence (see [29–31, 43, 45, 46, 74, 85]). Note that the equivalence classes of $\approx_{mar}$ correspond to the reachable markings of Σ.
- $C' \approx_{code} C''$ if $Mark(C') = Mark(C'')$ and $Code(C') = Code(C'')$, where $Code$ is the signal coding function. Such an equivalence is used in [86] for unfolding Signal Transition Graphs (STGs) specifying asynchronous circuits (see Chapter 8).
- $C' \approx_{sym} C''$ if $Mark(C')$ and $Mark(C'')$ are symmetric markings (according to some suitable notion of symmetry, see, e.g., [24, 49]). This equivalence is the basis of the approach exploiting symmetries to reduce the size of the prefix, described in [24].

For an equivalence relation $\approx$, we denote by $\mathfrak{R}_{\approx}^{fin} \stackrel{\text{df}}{=} \mathcal{C}_{fin}/_{\approx}$ the set of its equivalence classes, and by $\mathfrak{R}_{\approx}^{loc} \stackrel{\text{df}}{=} \mathcal{C}_{loc}/_{\approx}$ the set of the equivalence classes of the restriction of $\approx$ on the set of local configurations. We will also denote by $\Theta_{ERV} \stackrel{\text{df}}{=} (\approx_{mar}, \lhd_{erv}, \{\mathcal{C}_{loc}\}_{e \in E})$ the cutting context corresponding to the framework used in [30, 31, 85].

We will write $e \lhd f$ whenever $[e] \lhd [f]$. Clearly, $\lhd$ is a well-founded partial order on the set of events. Hence, we can use Noetherian induction (see Section 1.2), i.e., it suffices to define or prove something for an event under the assumption that it has already been defined or proved for all its $\lhd$-predecessors.

Proposition 2.3. *Let e and f be two events, and C be a finite configuration.*

1. *If $f \prec e$ then $f \lhd e$.*
2. *If $f \in C$ and $C \lhd [e]$ then $f \lhd e$.*

Proof. The first part follows from the fact that $f \prec e$ implies $[f] \subset [e]$, and so, by Definition 2.2(2), $[f] \lhd [e]$. To show the second part, we observe that $f \in C$ implies $[f] \subseteq C$, i.e., either $[f] = C$ and $f \lhd e$ trivially holds, or, by Definition 2.2(2), $[f] \lhd C$, and thus $[f] \lhd [e]$ by the transitivity of $\lhd$. $\square$

In the rest of this chapter, we assume that the cutting context Θ is fixed.

2.5.2 Completeness of branching processes

We now introduce a new notion of completeness for branching processes.

Definition 2.4. A branching process π is *complete w.r.t. a set E_{cut}* of events of Unf_{Σ}^{max} if the following hold:

1. If $C \in \mathcal{C}_{fin}$, then there is $C' \in \mathcal{C}_{fin}^{\pi}$ such that $C' \cap E_{cut} = \emptyset$ and $C \approx C'$.
2. If $C \in \mathcal{C}_{fin}^{\pi}$ is such that $C \cap E_{cut} = \emptyset$, and e is an event such that $C \oplus \{e\} \in \mathcal{C}_{fin}$, then $C \oplus \{e\} \in \mathcal{C}_{fin}^{\pi}$.

A branching process π is *complete* if it is complete w.r.t. some set E_{cut}. $\diamond$

Note that, in general, π remains complete after removing all events e for which $\langle e \rangle \cap E_{cut} \neq \emptyset$; i.e., without affecting the completeness, one can truncate a complete prefix so that the events from E_{cut} (usually referred to as *cut-off* events) will be either maximal events of the prefix or not in the prefix at all. Note also that the last definition depends only on the equivalence $\approx$, and not on the other components of the cutting context.

For the relation $\approx_{mar}$, each reachable marking is represented by a configuration in $\mathcal{C}_{fin}$ and, hence, also by a configuration in $\mathcal{C}_{fin}^{\pi}$, provided that π is complete (see Definition 2.4). This is what is usually expected from a correct prefix. But even in this special case, the proposed notion of completeness differs from that

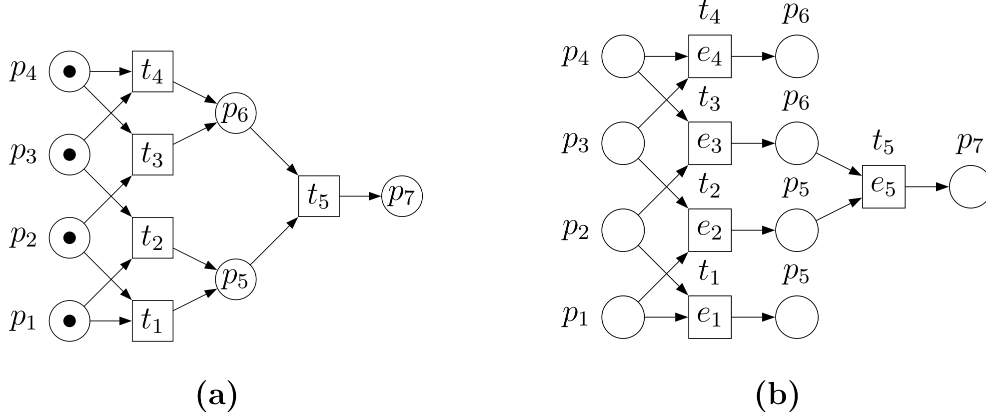

Figure 2.1: A Petri net **(a)** and one of its branching processes **(b)**, which is complete w.r.t. the definition used in [30,31], but not w.r.t. Definition 2.4. Note that the configuration $\{e_1, e_4\}$ does not preserve firings and introduces a fake deadlock. In order to make this prefix complete w.r.t. Definition 2.4 one has to add another instance of t_5, consuming the conditions produced by e_1 and e_4.

presented in [30,31,85], since it requires *all* configurations in $\mathcal{C}^{\pi}_{\mathit{fin}}$ containing no events from E_{cut} to preserve all transition firings, rather than the *existence* of a configuration preserving all firings. Figure 2.1 and Section 2.2 justify why such a stronger property is desirable. One can easily prove that this notion is strictly stronger than the one considered in [30, 31], i.e., that the completeness in the sense of Definition 2.4 implies the completeness in the sense of [30, 31], but not vice versa. However, it should be noted that the proof of completeness in [30,31] almost gives the stronger notion; we have adopted it (see Proposition 2.9) with relatively few modifications.

As an example, consider the net system in Figure 1.4. If $\approx$ is taken to be $\approx_{mar}$ then the prefix in Figure 1.4(b) is not complete w.r.t. any set E_{cut} for the following reason. The reachable marking $\{p_1, p_7\}$ is only represented by the local configurations of e_6 and e_8; for completeness, at least one of these events would not be in E_{cut}, but then its local configuration would have an extension with a t_1-labelled event, which is not the case. In contrast, the prefix in Figure 1.4(c) is complete w.r.t. the set $E_{cut} = \{e_5, e_{16}, e_{17}\}$.[3] Notice that the events e_8, e_9, e_{13}–e_{15}, e_{18}, and e_{19} can be removed from the prefix without affecting its completeness.

[3] This choice of E_{cut} is not unique: one could have chosen, e.g., $E_{cut} = \{e_4, e_{18}, e_{19}\}$.

2.6 Canonical prefix

In this section, we develop the central results of this chapter. First, we show that cut-off events can be defined without resorting to any algorithmic argument. This yields a definition of the canonical prefix, and we then prove several of its relevant properties.

2.6.1 Static cut-off events

In [30,31], the notion of a cut-off event was considered as algorithm-specific, and was given w.r.t. the already built part of a prefix. Now we define cut-off events w.r.t. the whole unfolding instead, so that it will be independent of an algorithm (hence the term 'static'), together with *feasible* events, which are precisely those events whose causal predecessors are not cut-off events, and as such must be included in the prefix determined by the static cut-off events.

Definition 2.5. The set of *feasible* events, denoted by fsble_Θ, and the set of *static cut-off* events, denoted by cut_Θ, are two sets of events of $\mathit{Unf}_\Sigma^{max}$ defined inductively, in the following way:

1. An event e is a feasible event if $\langle e \rangle \cap \mathit{cut}_\Theta = \emptyset$.

2. An event e is a static cut-off event if it is feasible, and there is a configuration $C \in \mathcal{C}_e$ such that $C \subseteq \mathit{fsble}_\Theta \setminus \mathit{cut}_\Theta$, $C \approx [e]$, and $C \lhd [e]$. In what follows, any C satisfying these conditions will be called a *corresponding* configuration of e. $\diamond$

Note that fsble_Θ and cut_Θ are well-defined sets due to Noetherian induction (see Section 1.2). Indeed, when considering an event e, by the well-foundedness of $\lhd$ and Proposition 2.3(1), one can assume that for the events in $\langle e \rangle$ it has already been decided whether they are in fsble_Θ or in cut_Θ. And, by Proposition 2.3(2), the same holds for the events in any configuration C satisfying $C \lhd [e]$. Therefore, the *status* of each event (whether it is feasible or not, and whether it is a static cut-off event or not) depends only on its $\lhd$-predecessors.

The above definition implies that $\bot \in \mathit{fsble}_\Theta$ since $\langle \bot \rangle = \emptyset$. Furthermore, $\bot \notin \mathit{cut}_\Theta$, since $\bot$ cannot have a corresponding configuration. Indeed, $[\bot] = \{\bot\}$ is the smallest (w.r.t. $\subseteq$) configuration, and so, by Definition 2.2(2), it is also $\lhd$-minimal.

Note that the structure of adequate orders can be quite complicated, in particular it is possible for an event to have infinitely many $\lhd$-predecessors (see Figure 2.2). This means that there might not exist a one-to-one correspondence $F : E \to \mathbb{N}$ between the events of a branching process and natural numbers such that $e' \lhd e''$ iff $F(e') < F(e'')$. Thus the standard induction on natural numbers is not always applicable, which justifies the use of Noetherian induction in Definition 2.5.

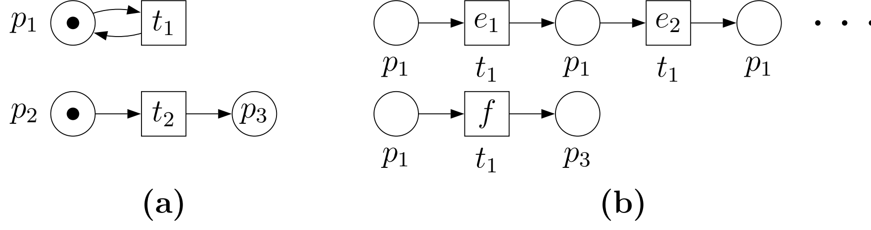

Figure 2.2: A Petri net **(a)** and its unfolding **(b)**. If the adequate order is defined as $C' \lhd C'' \iff C' \subset C'' \vee f \in C'' \setminus C'$ then f has infinitely many $\lhd$-predecessors.

Remark 2.6. A naïve attempt to define an algorithm-independent notion of a cut-off event as an event e for which there is a configuration $C \in \mathcal{C}_e$ such that $C \approx [e]$ and $C \lhd [e]$ generally fails for non-saturated cutting contexts, e.g., when (as it is often the case in practice) only local configurations can be used as cut-off correspondents. Indeed, a corresponding local configuration C of a cut-off event e defined in this way may contain another cut-off event. Though in this case Unf_Σ^{max} contains another corresponding configuration $C' \approx C$ with no cut-off events and such that $C' \lhd C$, C' is not necessarily local (see Figure 2.3).

The proposed approach, though slightly more complicated, allows one to deal uniformly with arbitrary cutting contexts. Moreover, it coincides with the described naïve approach when Θ is saturated. $\diamond$

Proposition 2.7. *Let e be an event of Unf_Σ^{max}.*

1. *$e \in fsble_\Theta$ iff $\langle e \rangle \subseteq fsble_\Theta \setminus cut_\Theta$.*

2. *$e \in cut_\Theta$ implies $e \in fsble_\Theta$.*

Proof. The 'if' part of the first clause follows directly from Definition 2.5(1). To prove the 'only if' part, suppose that $g \notin fsble_\Theta \setminus cut_\Theta$ for some $g \prec e$. Since $e \in fsble_\Theta$ and thus $\langle e \rangle \cap cut_\Theta = \emptyset$ by Definition 2.5(1), $g \notin fsble_\Theta$. Therefore, by Definition 2.5(1), $\langle g \rangle \cap cut_\Theta \neq \emptyset$, and so $\langle e \rangle \cap cut_\Theta \neq \emptyset$, a contradiction. Hence $\langle e \rangle \subseteq fsble_\Theta \setminus cut_\Theta$.

The second clause follows directly from Definition 2.5(2). $\square$

2.6.2 Canonical prefix and its properties

Once we have defined the feasible events, the notion of the canonical prefix arises quite naturally, after observing that the 'only if' part of Proposition 2.7(1) implies that $fsble_\Theta$ is a downward-closed set of events.

Definition 2.8 (Canonical Prefix). The branching process Unf_Σ^Θ induced by the set of events $fsble_\Theta$ is called the *canonical prefix* of Unf_Σ^{max}. $\diamond$

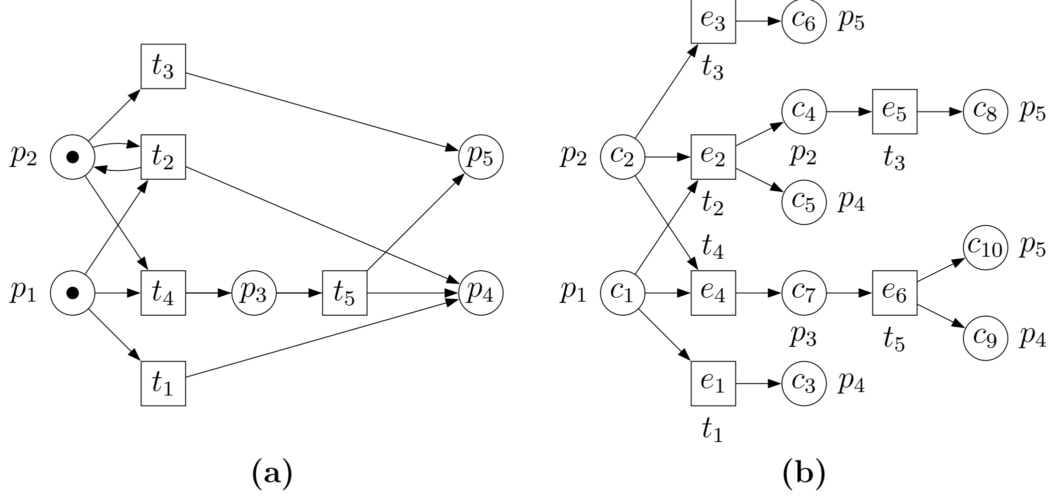

Figure 2.3: A Petri net **(a)** and its unfolding **(b)**. Assuming the cutting context Θ_{ERV}, e_6 is a cut-off event according to the naïve definition given in Remark 2.6, with a corresponding configuration $C \stackrel{\text{df}}{=} [e_5]$. However, C contains another cut-off event, viz. e_2 (with $[e_1]$ as a corresponding configuration), and thus is not in the prefix. Though the unfolding contains another corresponding configuration $C' \stackrel{\text{df}}{=} \{e_1, e_3\}$ with no cut-off events and such that $C' {\approx_{mar}} C {\approx_{mar}} [e_6]$ and $C' \lhd_{erv} C \lhd_{erv} [e_6]$, C' is not local.

Note that Unf_Σ^Θ is uniquely determined by the cutting context Θ.

In what follows, we prove several fundamental properties of Unf_Σ^Θ. Note that, unlike those given in [30, 31], the adduced proofs are not algorithm-specific.

Proposition 2.9 (Completeness). *Unf_Σ^Θ is complete w.r.t. $E_{cut} = cut_\Theta$.*[4]

Proof. (Compare the proof of Proposition 4.9 in [31]).

Let $\pi \stackrel{\text{df}}{=} Unf_\Sigma^\Theta$. To show Definition 2.4(1), suppose that $C \in \mathcal{C}_{fin}$. Let $C' \in \mathcal{C}_{fin}$ be $\lhd$-minimal among the configurations satisfying $C' \approx C$. Note that C' exists since, by Definition 2.2(2), $\lhd$ is well-founded. Suppose that $C' \cap cut_\Theta \neq \emptyset$, i.e., there is an event $e \in C' \cap cut_\Theta$. Then $C' = [e] \oplus E'$, for some finite suffix E' of $[e]$. Let C_e be a corresponding configuration of e. Since $C_e \approx [e]$ and $C_e \lhd [e]$, by Definition 2.2(3), there exists a suffix E'' of C_e such that $C_e \oplus E'' \approx [e] \oplus E'$ and $C_e \oplus E'' \lhd [e] \oplus E'$, i.e., $C_e \oplus E'' \approx C' \approx C$ and $C_e \oplus E'' \lhd C'$. Since $\lhd$ is strict, this contradicts the choice of C', and so $C' \cap cut_\Theta = \emptyset$. Therefore, by Definition 2.5(1), each event of C' is feasible, and so C' is a configuration of Unf_Σ^Θ containing no events from E_{cut}.

[4] Since the new notion of completeness is different from that given in [30, 31], we have to prove a stronger property than the one stated there, viz. that *all* configurations containing no cut-off events preserve *all* firings.

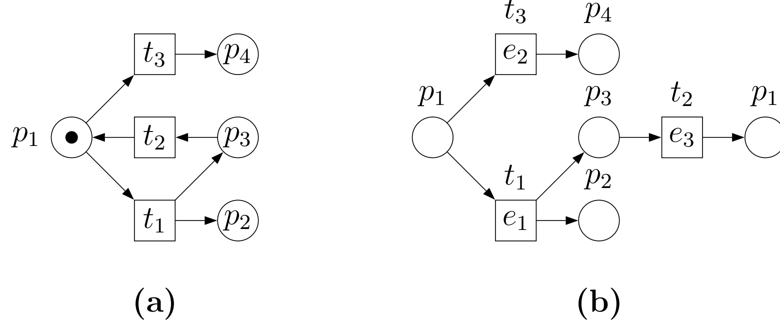

Figure 2.4: An unbounded net system **(a)** and its canonical prefix **(b)**. The cutting context is such that $C' \approx C'' \Leftrightarrow \mathit{Mark}(C') \cap \{p_1, p_3, p_4\} = \mathit{Mark}(C'') \cap \{p_1, p_3, p_4\}$ and $\{\bot\} \in \mathcal{C}_{e_3}$, and so e_3 is a static cut-off event.

To show Definition 2.4(2), suppose that $C \in \mathcal{C}^{\pi}_{\mathit{fin}}$, $C \cap E_{\mathit{cut}} = \emptyset$, and $C \oplus \{f\} \in \mathcal{C}_{\mathit{fin}}$. Then, by Proposition 2.7(1), $f \in \mathit{fsble}_{\Theta}$, and so $C \oplus \{f\} \in \mathcal{C}^{\pi}_{\mathit{fin}}$. □

Having proved that the canonical prefix is always complete, we now set out to analyze its finiteness. This property is, clearly, crucial if one intends to use such a prefix for model checking.

Proposition 2.10 (Finiteness I). *$\mathit{Unf}^{\Theta}_{\Sigma}$ is finite iff there is no infinite $\prec$-chain of feasible events in $\mathit{Unf}^{\mathit{max}}_{\Sigma}$.*

Proof. Follows directly from Definition 2.8 and Proposition 2.1. □

Thus, in order to guarantee that the canonical prefix is finite, one should choose the cutting context so that the $\mathcal{C}_e$'s contain enough configurations, and $\approx$ is coarse enough, to cut each infinite $\prec$-chain. It is interesting to observe that certain cutting contexts sometimes allow one to produce finite canonical prefixes even for unbounded net systems. Figure 2.4(a) shows a net system modelling a loop, where place p_2, used for counting the number of iterations, is unbounded. If $\approx$ ignores the value of this counter, it is possible to build the finite canonical prefix shown in Figure 2.4(b).

A necessary condition for the finiteness of the canonical prefix is the finiteness of the set $\mathfrak{R}^{\mathit{fin}}_{\approx}$ of the equivalence classes of $\approx$. Moreover, this becomes also a sufficient condition if Θ is dense. The following result provides quite a tight and practical indication whether $\mathit{Unf}^{\Theta}_{\Sigma}$ is finite or not.

Proposition 2.11 (Finiteness II).

1. *If $\mathfrak{R}^{\mathit{fin}}_{\approx}$ is finite and Θ is dense, then $\mathit{Unf}^{\Theta}_{\Sigma}$ is finite.*

2. *If $\mathfrak{R}^{\mathit{fin}}_{\approx}$ is infinite, then $\mathit{Unf}^{\Theta}_{\Sigma}$ is infinite.*

Proof. (1) (Compare the proof of Proposition 4.8 in [31].)

By Proposition 2.10, it is enough to show that there is no infinite $\prec$-chain of feasible events in Unf_{Σ}^{max}. To the contrary, suppose that such a chain $e_1 \prec e_2 \prec \cdots$ does exist. Since $\mathfrak{R}_{\approx}^{fin}$ is finite, there exist $i, j \in \mathbb{N}$ such that $i < j$ and $[e_i] \approx [e_j]$. Since $e_i \prec e_j$, we have $[e_i] \subset [e_j]$, and so $e_i \lhd e_j$ by Definition 2.2(2). Since Θ is dense, $[e_i] \in \mathcal{C}_{e_j}$, and since e_j is feasible, no event in $[e_i]$ belongs to cut_{Θ}. Thus, $e_j \in cut_{\Theta}$ and has no feasible causal successors, a contradiction.

(2) By Proposition 2.9, Unf_{Σ}^{Θ} is complete. Therefore, it contains at least one configuration from every equivalence class of $\approx$. Since a finite branching process contains only a finite number of configurations, Unf_{Σ}^{Θ} is infinite. $\square$

Corollary 2.12 (Finiteness III). *Let $\approx$ be either of $\approx_{mar}$, $\approx_{code}$, $\approx_{sym}$.*

1. *If Σ is bounded and Θ is dense, then Unf_{Σ}^{Θ} is finite.*
2. *If Σ is unbounded, then Unf_{Σ}^{Θ} is infinite.*

Proof. Follows from Proposition 2.11 and the fact that each of the three equivalences has a finite number of equivalence classes iff Σ is bounded. Indeed, since there are finitely many signals, the set $Code(\mathcal{C}_{fin})$ is finite, and so the set of combined states $\{(Mark(C), Code(C)) \mid C \in \mathcal{C}_{fin}\}$, which is isomorphic to $\mathfrak{R}_{\approx_{code}}^{fin}$, is finite iff the set $\mathcal{RM}(\Sigma)$ of reachable markings is finite. For $\approx_{sym}$, $|\mathfrak{R}_{\approx_{sym}}^{fin}| \leq |\mathfrak{R}_{\approx_{mar}}^{fin}|$, and so $\mathfrak{R}_{\approx_{sym}}^{fin}$ is finite if so is $\mathfrak{R}_{\approx_{mar}}^{fin}$. Moreover, since $C \approx_{sym} C'$ implies $|Mark(C)| = |Mark(C')|$, $\mathfrak{R}_{\approx_{sym}}^{fin}$ is infinite whenever $\mathfrak{R}_{\approx_{mar}}^{fin}$ is infinite. $\square$

In the important special case of a total adequate order, one can also derive an upper bound on the number of non-cut-off events in Unf_{Σ}^{Θ}. A specialized version of the next result (for $\Theta = \Theta_{ERV}$) was proven in [30,31,85] for the prefix generated by the unfolding algorithm presented there.

Proposition 2.13 (Upper Bound). *Suppose that $\lhd$ is total and:*

- *The set $\mathfrak{R}_{\approx}^{loc}$ is finite.*
- *For every $\mathcal{R} \in \mathfrak{R}_{\approx}^{loc}$, there is an integer $\gamma_{\mathcal{R}} > 0$ such that, for every chain $e_1 \lhd e_2 \lhd \cdots \lhd e_{\gamma_{\mathcal{R}}}$ of feasible events whose local configurations belong to $\mathcal{R}$, there is at least one $i \leq \gamma_{\mathcal{R}}$ such that $[e_i] \in \bigcap_{[e] \in \mathcal{R}} \mathcal{C}_e$.*

Then

$$|\, fsble_{\Theta} \setminus cut_{\Theta} \,| \leq \sum_{\mathcal{R} \in \mathfrak{R}_{\approx}^{loc}} \gamma_{\mathcal{R}} \,. \tag{2.1}$$

Proof. First, we show that every equivalence class $\mathcal{R} \in \mathfrak{R}_{\approx}^{loc}$ contains at most $\gamma_{\mathcal{R}}$ feasible non-cut-off events. To the contrary, suppose that $e_1, \ldots, e_{\gamma_{\mathcal{R}}+1} \in fsble_{\Theta} \setminus cut_{\Theta}$ are distinct events whose local configurations belong to the same equivalence class $\mathcal{R}$ of $\approx$. Since $\lhd$ is total, we may assume, without loss of generality, that $e_1 \lhd \ldots \lhd e_{\gamma_{\mathcal{R}}+1}$. Hence, for at least one $i \leq \gamma_{\mathcal{R}}$, $[e_i] \in \bigcap_{[e] \in \mathcal{R}} \mathcal{C}_e \subseteq \mathcal{C}_{e_{\gamma_{\mathcal{R}}+1}}$. To

summarize, $[e_i] \approx [e_{\gamma_\mathcal{R}+1}]$ (because these two events are in the same equivalence class), $e_i \lhd e_{\gamma_\mathcal{R}+1}$, and $[e_i] \in \mathcal{C}_{e_{\gamma_\mathcal{R}+1}}$, i.e., $e_{\gamma_\mathcal{R}+1} \in \mathit{cut}_\Theta$ by Definition 2.5(2), a contradiction. Thus every equivalence class $\mathcal{R} \in \mathfrak{R}^{loc}_{\approx}$ contains at most $\gamma_\mathcal{R}$ feasible non-cut-off events, and (2.1) holds. □

Note that if Θ is dense, then $\gamma_\mathcal{R} = 1$ for every $\mathcal{R} \in \mathfrak{R}^{loc}_{\approx}$, and

$$|\, \mathit{fsble}_\Theta \setminus \mathit{cut}_\Theta \,| \leq |\mathfrak{R}^{loc}_{\approx}| \leq |\mathfrak{R}^{fin}_{\approx}| \,.$$

In the case $\Theta = \Theta_{ERV}$, the upper bound on the number of non-cut-off events in the prefix derived in [31] can now be obtained as follows. Since the reachable markings of Σ correspond to the equivalence classes of $\approx_{mar}$, $|\, \mathit{fsble}_\Theta \setminus \mathit{cut}_\Theta \,| \leq |\mathcal{RM}(\Sigma)|$ by the above formula. Using Proposition 2.13, one can easily derive the following upper bounds for the remaining two equivalences considered in this chapter (in each case, we assume that Θ is dense):

- $|\mathfrak{R}^{fin}_{\approx_{code}}| = |\{(\mathit{Mark}(C), \mathit{Code}(C))\}_{C \in \mathcal{C}_{fin}}| \leq |\mathit{Mark}(\mathcal{C}_{fin})| \cdot |\mathit{Code}(\mathcal{C}_{fin})| \leq |\mathcal{RM}(\Sigma)| \cdot 2^{|Z|}$, where Z is the set of STG's signals. This bound can be improved to $|\mathcal{RM}(\Sigma)|$ if certain 'pathological' STGs are excluded (see Chapter 8).
- $|\mathfrak{R}^{fin}_{\approx_{sym}}| \leq |\mathfrak{R}^{fin}_{\approx_{mar}}| = |\mathcal{RM}(\Sigma)|$.

Note that these upper bounds are rather pessimistic, particularly because we bound $|\mathfrak{R}^{loc}_{\approx}|$ by $|\mathfrak{R}^{fin}_{\approx}|$. In practice, the set $\mathfrak{R}^{fin}_{\approx}$ is usually exponentially larger than $\mathfrak{R}^{loc}_{\approx}$, and so prefixes are often exponentially smaller than reachability graphs.

2.7 Unfolding algorithms

We now show that suitable modifications of existing unfolding algorithms generate the canonical prefix defined in the previous section. In particular, this is the case for the algorithm presented in [30,31,85] and the parallel unfolding algorithm proposed in [45, 46]. The latter will also be described in detail in Chapter 5.

2.7.1 ERV unfolding algorithm

The unfolding algorithm presented in [30, 31] can be expressed as shown in Figure 2.5. We will call it the *basic algorithm*. In the present setup, it is parameterized by a cutting context Θ. It is assumed that the function PotExt(*Pref*) finds the set of possible extensions of a branching process *Pref*, according to Definition 1.3.

When $\lhd$ is a total order, the algorithm in Figure 2.5 is deterministic, and thus always yields the same result for a given net system. A rather surprising fact is that this is also the case for an arbitrary adequate order for which the

input : $\Sigma = (N, M_0)$ — a net system
output : *Pref* — the canonical prefix of Σ's unfolding (if it is finite)

Pref $\leftarrow$ the empty branching process
add instances of the places from M_0 to *Pref*
$pe \leftarrow$ PotExt(*Pref*)
$cut_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  choose $e \in \min_{\lhd} pe$
  if $[e] \cap cut_off = \emptyset$
  then
    $Pref \leftarrow Pref \oplus \{e\}$
    $pe \leftarrow$ PotExt(*Pref*)
    if e is a cut-off event of *Pref* **then** $cut_off \leftarrow cut_off \cup \{e\}$
  else $pe \leftarrow pe \setminus \{e\}$
$Pref \leftarrow Pref \oplus cut_off$

Note: e is a cut-off event of *Pref* if there is $C \in \mathcal{C}_e$ such that
the events of C belong to *Pref* but not to *cut_off*, $C \approx [e]$, and $C \lhd [e]$.

Figure 2.5: The unfolding algorithm presented in [30,31].

algorithm is, in general, non-deterministic. This fact was proven in [45,46] in a rather complicated way, by comparing two runs of the algorithm. The theory of canonical prefixes developed in the previous section can provide a more elegant proof (given later in this section), by showing that the algorithm always generates the canonical prefix.

2.7.2 Unfolding with slices

An unfolding algorithm which admits efficient parallelization (proposed in [45,46] and described in more detail in Chapter 5) is shown in Figure 2.6. We will call it the *slicing algorithm*. When compared to the basic algorithm, it has the following modifications in its main loop. A set of events $Sl \in$ Slices(pe), called a *slice* of the current set of possible extensions, is chosen on each iteration and processed as a whole, without taking or adding any other events from or to pe.

Definition 2.14 (Slice). A *slice* Sl is a non-empty subset of pe such that, for every event $e \in Sl$ and every event $f \lhd e$ of Unf_{Σ}^{max}, $f \notin pe \setminus Sl$ and $pe \cap \langle f \rangle = \emptyset$.

In particular, if $f \in pe$ and $f \lhd e$ for some $e \in Sl$, then $f \in Sl$. The set Slices(pe) is chosen so that it is non-empty whenever pe is non-empty. Note that this algorithm, in general, exhibits more non-determinism than the basic one (it may be non-deterministic even if the order $\lhd$ is total).

```
input : Σ = (N, M_0) — a net system
output : Pref — the canonical prefix of Σ's unfolding (if it is finite)

Pref ← the empty branching process
add instances of the places from M_0 to Pref
pe ← PotExt(Pref)
cut_off ← ∅
while pe ≠ ∅ do
   choose Sl ∈ Slices(pe)
   if ∃e ∈ Sl : [e] ∩ cut_off = ∅
   then
      for all e ∈ Sl in any order refining ◁ do
         if [e] ∩ cut_off = ∅
         then
            Pref ← Pref ⊕ {e}
            if e is a cut-off event of Pref then cut_off ← cut_off ∪ {e}
      pe ← PotExt(Pref)
   else pe ← pe \ Sl

Pref ← Pref ⊕ cut_off
```

Note: e is a cut-off event of *Pref* if there is $C \in \mathcal{C}_e$ such that the events of C belong to *Pref* but not to *cut_off*, $C \approx [e]$, and $C \lhd [e]$.

Figure 2.6: Unfolding algorithm with slices.

It was proven (in a very complicated way) in [45] that the unfolding algorithms shown in Figures 2.5 and 2.6 are equivalent, in the sense that prefixes produced by arbitrary runs of these algorithms are isomorphic. Here, we prove this result by showing that arbitrary runs of these algorithms generate the canonical prefix.

Since the basic algorithm can be obtained as a special case of the slicing one, by setting $\textsc{Slices}(pe) \overset{\text{df}}{=} \{\{e\} \mid e \in \min_{\lhd} pe\}$ (compare [45, 46]), the proofs of Lemmas 2.15–2.18 below are given only for the slicing algorithm.

Lemma 2.15. *Consider the state of the algorithm in Figure 2.6 before adding an event e to Pref. If $cut_off \subseteq cut_\Theta$, $g \in fsble_\Theta$ and $g \lhd e$, then g is in Pref.*

Proof. Let $Pref'$ be the state of the variable *Pref* at the moment when a slice Sl was chosen in the current iteration of the main loop of the algorithm.

Suppose that $g \in fsble_\Theta$ is such that $g \lhd e$ and g is not in *Pref*. Consider the set $G \overset{\text{df}}{=} \{h \in [g] \mid h \text{ is not in } Pref'\}$. Clearly, $G \neq \emptyset$ since $g \in G$ (as g is not in *Pref* and the algorithm never removes events from the prefix being constructed). Thus there exists $f \in \min_{\prec} G$. By Proposition 2.3(2), $f \lhd e$. Moreover, $f \in pe$ because all its causal predecessors are in $Pref'$ by the choice of f.

By Definition 2.14 and the facts that $e \in Sl$, $f \lhd e$ and $f \in pe$, we have that $f \in Sl$. Moreover, if $f \neq g$ then $f \prec g$, contradicting Definition 2.14, since $e \in Sl$, $g \lhd e$, and $pe \cap \langle g \rangle \neq \emptyset$ (because $f \in pe \cap \langle g \rangle$). Therefore, $g = f \in Sl$. Since $g \lhd e$, it was processed before e in the **for all** loop of the algorithm. By $g \in \mathit{fsble}_\Theta$, $\mathit{cut_off} \subseteq \mathit{cut}_\Theta$ and Proposition 2.7(1), we obtain that $\langle g \rangle \cap \mathit{cut_off} \subseteq \langle g \rangle \cap \mathit{cut}_\Theta = \emptyset$. Moreover, $g \notin \mathit{cut_off}$ when g is being processed. Therefore, g has been added to *Pref* before the processing of e, a contradiction. □

Lemma 2.16 (Soundness). *If the algorithm in Figure 2.6 adds an event e to Pref, then $e \in \mathit{fsble}_\Theta$. Moreover, such an event e is added to cut_off iff $e \in \mathit{cut}_\Theta$.*

Proof. By induction on the number of events added before e. When e is being added to *Pref*, the condition $[e] \cap \mathit{cut_off} = \emptyset$ is satisfied. Since the events in $\langle e \rangle$ have been added before, $\langle e \rangle \cap \mathit{cut}_\Theta = \emptyset$ by induction, and so $e \in \mathit{fsble}_\Theta$.

If e is then added to *cut_off* due to some corresponding configuration $C \lhd [e]$, then the events of C belong to *Pref* but not to cut_Θ. Hence, by induction, $C \subseteq \mathit{fsble}_\Theta \setminus \mathit{cut}_\Theta$. Thus, $e \in \mathit{cut}_\Theta$.

Now assume an event e added to *Pref* is in cut_Θ with a corresponding configuration C. Since, by Proposition 2.3(2), $g \in C \lhd [e]$ implies $g \lhd e$, and by the fact that $\mathit{cut_off} \subseteq \mathit{cut}_\Theta$ before e was added to *Pref* (by induction), each $g \in C$ has already been added to *Pref* by Lemma 2.15. Furthermore, $g \notin \mathit{cut_off}$, by $C \cap \mathit{cut}_\Theta = \emptyset$ and the induction hypothesis. Therefore, the algorithm will add e to *cut_off*. □

Lemma 2.17 (Termination). *If $\mathit{Unf}_\Sigma^\Theta$ is finite, then the algorithm in Figure 2.6 terminates in a finite number of steps.*

Proof. The only time when events are added to *pe* is the call to PotExt. Such a call returns only a finite set of possible extensions, and so *pe* is always finite. In the body of the **while** loop, non-empty (by Definition 2.14) slices are removed from *pe*. This is repeated until a new event is added to *Pref*, which, by the assumption and Lemma 2.16, can happen only finitely many times, or until *pe* becomes empty and the algorithm terminates. □

Lemma 2.18 (Completeness). *Let $e \in \mathit{fsble}_\Theta$. If the algorithm in Figure 2.6 terminates yielding a prefix Pref, then e is an event of Pref.*

Proof. Suppose that e is a $\lhd$-minimal event of fsble_Θ which is not in *Pref* at termination. All causal predecessors of e are in *Pref*, but, by Lemma 2.16, not in *cut_off*. Thus, e was in *pe* after possible extensions were computed for the last time; the condition $[e] \cap \mathit{cut_off} = \emptyset$ holds at termination, and thus has been holding before. Therefore, since *pe* is empty at termination, e was added to *Pref*, a contradiction. □

Proposition 2.19 (Correctness). *If $\mathit{Unf}_\Sigma^\Theta$ is finite, then the algorithms in Figures 2.5 and 2.6 generate $\mathit{Unf}_\Sigma^\Theta$ in a finite number of steps.*

Proof. Follows directly from Lemmas 2.16, 2.17, and 2.18, and the fact that the basic algorithm is a special case of the slicing one. □

In particular, this result implies the completeness of prefixes produced by the basic and slicing algorithms w.r.t. Definition 2.4 (and thus w.r.t. the weaker definition given in [30,31]), and the fact that arbitrary runs of these non-deterministic algorithms always yield the same result, viz. the canonical prefix.

2.8 Conclusions

We presented a general framework for truncating Petri net unfoldings. It provides a powerful tool for dealing with different variants of the unfolding technique, in a flexible and uniform way. In particular, by finely tuning the cutting contexts, one can build prefixes which better suit a particular model checking problem. A fundamental result is that, for an arbitrary Petri net and a cutting context, there exists a 'special' canonical prefix of its unfolding, which can be defined without resorting to any algorithmic argument.

We introduced a new, stronger notion of completeness of a branching process, which was implicitly assumed by many existing model checking algorithms employing unfoldings (see Figure 2.1 and the explanation in Section 2.2). We have shown that the canonical prefix is complete w.r.t. this notion, and that it is exactly the prefix generated by arbitrary runs of the non-deterministic unfolding algorithms presented in [30, 31, 45, 46, 85]. This gives a new correctness proof for the unfolding algorithms presented there, which is much simpler in the case of the slicing algorithm developed in [45, 46]. As a result, relevant model checking tools can now make stronger assumptions about the properties of the prefixes they use. In particular, they can safely assume that each configuration containing no cut-off events preserves all firings.

Finally, we proposed conditions for the finiteness of the canonical prefix, and presented criteria allowing bounds to be placed on its size, which should help in choosing problem-specific cutting contexts. It is worth noting that in order to deal with the finiteness problem we proved a version of König's Lemma for branching processes of (possibly unbounded) Petri nets.

We believe that the results contained in this chapter, on the one hand, will help to understand better the issues relating to prefixes of Petri net unfoldings, and, on the other hand, will facilitate the design of efficient model checking tools.

Chapter 3

Test Bench

Implementations of unfolding algorithms are usually quite intricate and error prone, and as such require extensive testing. But testing itself is a complicated problem, since it is not trivial to check whether a generated prefix is correct. In this chapter, we develop an approach to testing unfolding algorithms and describe the test cases. These test cases are used in Chapters 4, 5, and 7 to check the correctness of the algorithms described there, and to evaluate their performance (Chapters 6 and 8 use their own specific benchmarks).

3.1 Testing Unfolding Algorithms

Due to the canonicity of the prefix for a given cutting context (see Chapter 2), any two correct implementations of (possibly non-deterministic) unfolding algorithms produce the same result. Therefore, it is possible to experimentally confirm the correctness of an implementation of an unfolding algorithm, by checking that the resulting prefixes are isomorphic to those generated by an independently developed implementation. For this purpose, we used the ERVUNFOLD 4.5.1 unfolder by Stefan Römer (see [29–31, 85]).

Though the prefixes must be isomorphic, they are not necessarily identical, since their events and conditions might be generated in different orders. Therefore, a special utility for 'sorting' files containing prefixes of safe net systems was developed, so that if two prefixes were isomorphic then after 'sorting' the corresponding files become identical. Assuming that the prefix is represented as a list of events and a list of conditions, and to each node a list of nodes in its preset and a list of nodes in its postset are attached, the 'sorting' algorithm can be described as follows:

1. Separate cut-off events, pushing them to the end of the list of events.
2. Sort the other events in the list according to $\lhd_{erv}$.
3. Separate the conditions occurring in the postsets of cut-off events, pushing them to the end of the list of conditions.

4. Sort the other conditions according to the following ordering: $c' \lessdot c''$ if $e' \lhd_{erv} e''$, or $e' = e''$ and $h(c') \ll h(c'')$, where $\{e'\} = {}^\bullet c'$, $\{e''\} = {}^\bullet c''$, and $\ll$ is an arbitrary total order on the places of the original net system (e.g., the size-lexicographical ordering on their names).
   Note that e and e' are non-cut-off events, and that the non-cut-off events of the prefix have already been sorted according to $\lhd_{erv}$ by this step.

5. Sort the presets of the events (including the cut-off events) according to $\lessdot$.

6. Sort the cut-off events according to the following ordering: $e' \lessdot e''$ if ${}^\bullet e' \lessdot_{sl} {}^\bullet e''$, or ${}^\bullet e' = {}^\bullet e''$ and $h(e') \ll h(e'')$, where $\lessdot_{sl}$ is the size-lexicographical order, built upon $\lessdot$, and $\ll$ is an arbitrary total order on the set of the transitions of the original net system (e.g., the size-lexicographical ordering on their names).
   Note that the conditions which can appear in the presets of the events have already been sorted by this step.

7. Sort the part of the list of conditions containing the conditions occurring in the postsets of cut-off events according to $\lessdot$.
   Note that all events have already been sorted by this step.

8. Sort the postsets of the events (including the cut-offs) according to the $\lessdot$ ordering.
   Note that all conditions have already been sorted by this step.

3.2 Test cases

Here we describe low-level Petri net benchmarks, which are used in Chapters 4 and 5 to test the performance of unfolding algorithms described there. Moreover, the canonical prefixes of their unfoldings are used in Chapter 7 to test the integer programming model checking algorithm. All the experiments except those in Chapter 5 were conducted on a PC with a Pentium^{TM} III/500MHz processor and 128M RAM.

In most of experiments the popular set of benchmark examples collected by J.C. Corbett ([19]), K. McMillan, S. Melzer, S. Merkel, and S. Römer was attempted (many of the examples from this set were also used in [29, 40–44, 52–55, 77]). They are as follows:

ABP — Alternating Bit Protocol. A simple but often analyzed model with 6 tasks representing two users, a sender, a receiver, and two lossy channels.

BDS — Border Defense System. This example is the communication skeleton of a real Ada tasking program that simulates a border defense system. The example has 15 tasks, but the skeleton of each is relatively simple.

BUF(100) — Buffer of capacity 100 (with 2^{100} reachable states).

BYZ — Byzantine agreement protocol for 4 processors, one of which is faulty.

CYCLIC(n) — Milner's Cyclic Scheduler. It uses n scheduler tasks to keep n customer tasks loosely synchronized.

DAC(n) — Divide and Conquer. A program modelling a divide and conquer computation by forking up to n solver tasks that proceed in parallel.

DME(n) — Distributed mutual exclusion asynchronous circuit with n cells.

DP(n), DPD(n), DPFM(n), and DPH(n) — Dining Philosophers. In addition to the standard version DP(n), which can deadlock, several versions of the problem where deadlock is prevented will also be analyzed.

In the dictionary version DPD(n), the deadlock is prevented by having the philosophers pass a dictionary around the table. The philosopher holding the dictionary cannot hold any forks.

In the version with a fork manager DPFM(n), philosophers pick up both forks simultaneously by rendezvous with a fork manager task, which records the state of all forks in lieu of the fork tasks.

Finally, in the version with a host DPH(n), there is an additional host task with which a philosopher must synchronize before attempting to acquire his forks. The host will allow at most $n-1$ philosophers to hold forks at any one time.

ELEV(n) — Elevators. A model of a controller for a building with n elevators, using tasks to model the behaviour of the elevators themselves. The size n version has $n+3$ tasks.

FTP(n) — File Transfer Program. A model of a program which services requests from n users to transfer files over a network. The size n version has $n+8$ tasks.

FURN(n) — Remote Furnace Program. This program manages temperature data collection for n furnaces. The size n version has $2n+6$ tasks.

GASQ(n), GASNQ(n) — Gas Station (queueing and non-queueing versions). This example models a self-service gas station. The model has one operator task, two pump tasks, and n customer tasks.

HART(n) — Hartstone Program. The communication skeleton of an Ada program in which one task starts and then stops n worker tasks.

KEY(n) — Keyboard Program. The communication skeleton of an Ada program that manages keyboard/screen interaction in a window manager. The program is scaled by making the number of customer tasks a parameter n. The size n version has $n+5$ tasks.

MMGT(n) — Distributed Memory Manager. The communication skeleton of an Ada program implementing the memory management scheme with n users. The size n version has $n + 4$ tasks.

OVER(n) — Overtake Protocol. An Ada version of an automated highway system overtake protocol for n cars comprising $2n + 1$ tasks.

Q — User Interface. A model of an RPC client/server-based user interface with 18 tasks that is used by several real applications.

RING(n) — Token Ring Mutual Exclusion Protocol. A model of a standard distributed mutual exclusion algorithm in which n user tasks synchronize access to a resource through n sever tasks that pass a token around a ring.

RW(n) — Readers and Writers. A model of a database that may be simultaneously accessed by any number of readers or a single writer. Each of the n reader tasks and n writer tasks must synchronize with a controller task before accessing and when finished accessing the database.

SENT(n) — Sensor Test Program. The communication skeleton of an Ada program that starts up n tasks to test sensors. The size n version has $n+4$ tasks.

SPEED — Speed Regulation Program. The communication skeleton of an Ada program with 10 tasks that monitor and regulate the speed of a car.

SYNC(n) — Readers/Writers Synchronization. A model of a scalable and bottleneck-free readers/writers synchronization algorithm for shared memory parallel machines.

A more detailed description of these examples can be found in [19, 77].

In order to test the algorithms on nets with larger presets (in particular, to test the procedure for generating the possible extensions of a prefix, described in Chapter 4), we have built a set of examples RND(m, n, k) in the following way. First, m loops consisting of n places and n transitions each were created; the first place of each loop was marked with one token. Then k additional transitions were added to this skeleton, so that each of them takes a token from a randomly chosen place in each loop and puts it back in another randomly chosen place of the same loop (thus, the net has mn transitions with presets of size 1 and k transitions with presets of size m). It is easy to see that the nets built in this way are safe. Moreover, we modelled (with help of A. Bystrov) the priority arbiter circuits based on dual-rail logic, described in [9]. Basically, a priority arbiter handles requests from several concurrent processes and decides (using some priority system) which request should be granted. We generated two series of examples: SPA(n) for n processes and linear priorities (i.e., among the processes sending requests, the one with the smallest number is granted first),

and SPA(m, n) for m groups and n processes in each group with the following priority function:

1. Group with the largest number of requests is handled first.
2. Among several groups with the same number of requests, the one having the smallest number is handled first.
3. Within a group, the process with the smallest number, sending a request, is granted first.

The priority function is in fact a boolean function of many variables, and so these series of examples have many transitions with relatively large presets.

Chapter 4

Computing Possible Extensions

As it was already mentioned in the Introduction, prefix-based model checking is typically done in two steps: (i) generating a finite and complete prefix, and (ii) checking a relevant property on the generated prefix. In view of recent very fast model checking algorithms employing unfoldings (e.g., the integer programming algorithm described in Chapter 7 and the one based on computing a stable model of a logic program from [40–42, 44]), the problem of efficiently building them becomes crucial. In fact, building a prefix is often the bottleneck of the prefix-based verification. [29–31, 85] address this issue, considerably improving the original McMillan's technique, but we think that certain important details of the unfolding algorithm proposed there should be further developed.

Though the complexity of constructing a prefix in the worst case is high (see [28, 42]), in practice there are many cases, when unfoldings can be built quite efficiently. [30, 31] mentioned that the slowest part of their unfolding algorithm was building possible extensions of the unfolding being constructed (this is, in fact, an NP-complete problem,[1] see [42]), but the question how actually to efficiently compute them was left open. [29, 85] suggests to keep concurrency relation (i.e., the *co* relation defined in Section 1.5) as a set of bit-vectors and provides a method of efficiently maintaining and using this data structure. This approach is quick for simple systems, but soon becomes intractable (the amount of memory to store this relation is proportional to the product of the numbers of conditions and events in the already built part of the prefix).

In this chapter, we describe another method of computing possible extensions, applicable to safe net systems and fully compatible with the concurrency relation. Essentially, we show how to find new transition instances to be inserted in the unfolding, not trying all the transitions one-by-one, but all at once, merging common parts of the work. Moreover, we provide some additional heuristics, helping to speed up the algorithm. Experiments demonstrate that the resulting

[1] The maximum size of a transition preset together with the size of the currently built part of the prefix must be parameters of the algorithm deciding whether the set of possible extensions is empty or not. If the maximum size of a transition preset is fixed, there exists a polynomial-time algorithm.

algorithms can achieve significant speedups if the transitions of the Petri net being unfolded have large presets.

This chapter is based on the results developed in [56, 57].

4.1 Employing the UpdatePotExt function

Almost all the steps of the unfolding algorithm in Figure 2.5 can be implemented quite efficiently. The only hard part is to calculate the set of possible extensions, PotExt(*Pref*), and we will concentrate on this task. As the decision version of the problem is NP-complete in the size of the already built part of the prefix ([42]), it is unlikely that we can achieve substantial improvements in the worst case for a single call to the PotExt subroutine. However, the following approaches can still be attempted: (i) using heuristics to reduce the cost of a single call; and (ii) merging the common parts of the work performed to insert individual instances of transitions. An excellent example of a method aimed at reducing the amount of work is the improvement proposed in [30, 31], where a total order on configurations is used to reduce both the size of the constructed complete prefix and the number of calls to PotExt. Another method is outlined in [29, 75], where the algorithm does not have to recompute all the possible extensions in each step: it suffices to update the set of possible extensions left from the previous call, by adding events consuming conditions from $e^\bullet$, where e is the last inserted event. Formally, the algorithm computes the (Pref, e)-extensions (rather than all possible extensions of *Pref*), according to Definition 1.3.

With this approach, the set *pe* in the algorithm in Figure 2.5 can be seen as a priority queue (with the events ordered according to the adequate order $\lhd$ on their local configurations) and implemented using, e.g., a binary heap.[2] The call to PotExt(*Pref*) in the body of the main loop of the algorithm is replaced by UpdatePotExt(Pref, e), which finds all the possible (Pref, e)-extensions. This also reduces the size of the NP-complete problems to be solved at least by one; in particular, in the important special case of binary synchronization, when the size of a transition's preset is at most 2, say ${}^\bullet t = \{h(c), p\}$, the problem is equivalent to finding the set $\{c' \in h^{-1}(p) \mid c' \ co\ c\}$, which can be efficiently computed (note that the problem becomes vacuous when $|{}^\bullet t| = 1$). Moreover, this approach leads to a further simplification in the algorithm of Figure 2.5.[3] Indeed, since now we can be sure that we do not compute any possible extension more than once, we do not have to add the cut-off events (and their postsets) into the prefix being built until the very end of the algorithm. Hence we can altogether avoid checking whether a configuration contains a cut-off event. Furthermore,

[2]The slicing algorithm shown in Figure 2.6 allows for a more efficient data structure for *pe*. It will be discussed in detail in Chapter 5.

[3]For the sake of simplicity, the discussion in this chapter concentrates on the basic unfolding algorithm, but all the results are directly applicable to the slicing algorithm. The only thing which needs to be done is to replace the call to PotExt in the main loop of the slicing algorithm by the call to UpdatePotExt, as will be discussed in Chapter 5.

input : $\Sigma = (N, M_0)$ — a safe net system
output : *Pref* — the canonical prefix of Σ's unfolding

Pref $\leftarrow$ the empty branching process
add instances of the places from M_0 to *Pref*
$pe \leftarrow$ PotExt(*Pref*)
$cut_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  choose $e \in \min_{\triangleleft} pe$
  $pe \leftarrow pe \setminus \{e\}$
  if e is a cut-off event of *Pref*
  then $cut_off \leftarrow cut_off \cup \{e\}$
  else
    $Pref \leftarrow Pref \oplus \{e\}$
    $pe \leftarrow pe \cup$ UpdatePotExt($Pref, e$)
$Pref \leftarrow Pref \oplus cut_off$

Note: e is a cut-off event of *Pref* if there is $C \in \mathcal{C}_e$ such that
the events of C belong to *Pref* but not to *cut_off*, $C \approx [e]$, and $C \triangleleft [e]$.

Figure 4.1: An improved version of the basic unfolding algorithm.

in the case of a dense cutting context and total adequate order $\triangleleft$, the check of the cut-off criterion can be simplified. Indeed, in such a case it is guaranteed that, on each iteration of the main loop of the algorithm, the event extracted from the queue is greater (w.r.t. $\triangleleft$) than any event in *Pref*, and so one can skip computing $\triangleleft$. In fact, one can implement this test as one look-up in a hash table containing the equivalence classes of $\approx$ for each local configuration of *Pref* (e.g., for $\approx_{mar}$ the hash table contains final markings of local configurations). The resulting unfolding algorithm is shown in Figure 4.1. At this point, we set out for optimizing the UpdatePotExt subroutine.

4.2 Reducing the number of candidates

Note that in Definition 1.3, e and (t, D) are not separated events, which basically suggests that any sufficient condition for being a pair of separated events may help in reducing the computational cost involved in calculating the set of (π, e)-extensions. In what follows, we identify two such cases.

In the pseudo-code given in [75], the conditions $c \in e^{\bullet}$ are inserted into the unfolding one by one, and the algorithm tries to insert new instances of transitions from $h(c)^{\bullet}$ with c in their presets. Such an approach can be improved as the algorithm is sub-optimal in the case when a transition t can consume

more then one condition from $e^\bullet$. Indeed, t is considered for insertion after each condition from $e^\bullet$ it can consume has been added; this may lead to a significant overhead when the size of t's preset is large. Therefore, it is better to insert into the unfolding the whole postset $e^\bullet$ at once, and use the following simple result, which essentially means that possible extensions being added consume as many conditions from $e^\bullet$ as possible (note that this results in an improvement each time when there is a transition $t \in (h(e)^\bullet)^\bullet$ such that its instance can consume more than one condition produced by e).

Proposition 4.1. *Let e and f be events in a branching process of a safe net system such that $f \in (e^\bullet)^\bullet$ and $h(e^\bullet \cap {}^\bullet f) \neq h(e)^\bullet \cap {}^\bullet h(f)$. Then e and f are separated.*

Proof. Since $h(e^\bullet \cap {}^\bullet f) \subseteq h(e)^\bullet \cap {}^\bullet h(f)$ always holds, there exists a place $p \in (h(e)^\bullet \cap {}^\bullet h(f)) \setminus h(e^\bullet \cap {}^\bullet f)$. By the definition of a branching process, there are distinct non-conflicting p-labelled conditions $c \in e^\bullet$ and $d \in {}^\bullet f$. Hence, as the net system is safe, c and d are not concurrent, i.e., $c \prec d$ or $d \prec c$ holds. Since the latter contradicts $f \in (e^\bullet)^\bullet$, we have $c \prec d$, and so e and f are separated. □

Corollary 4.2. *Let π be a branching process of a safe net system, e be an event of π, and (t, D) be a (π, e)-extension. Then $|e^\bullet \cap D| = |h(e)^\bullet \cap {}^\bullet t|$.*

Another way of reducing the number of calls to PotExt is to ignore some of the transitions from $(u^\bullet)^\bullet$, which the algorithm attempts to insert after a u-labelled event e. Indeed, in a safe net system, if the preset ${}^\bullet t$ of a transition $t \in (u^\bullet)^\bullet$ has non-empty intersection with ${}^\bullet u \setminus u^\bullet$, then t cannot be executed immediately after u. Therefore, in the unfolding procedure, an instance f of t cannot be inserted immediately after a u-labelled event e (though f *may* actually consume conditions produced by e, as shown in Figure 4.2; note that in such a case e and f are separated).

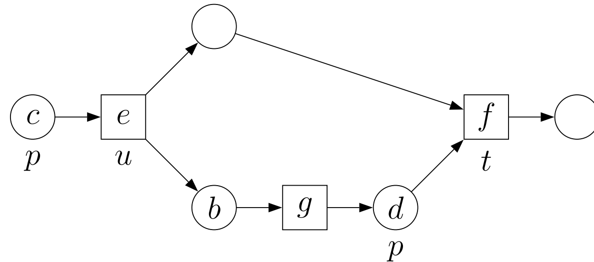

Figure 4.2: If ${}^\bullet t \cap ({}^\bullet u \setminus u^\bullet) \neq \emptyset$ then a t-labelled event f cannot be inserted immediately after a u-labelled event e, even though it can consume a condition produced by e.

Proposition 4.3. *Let e and f be events in the unfolding of a safe net system such that $f \in (e^\bullet)^\bullet$ and ${}^\bullet h(f) \cap ({}^\bullet h(e) \setminus h(e)^\bullet) \neq \emptyset$. Then e and f are separated.*

Proof. Let $p \in {}^\bullet h(f) \cap ({}^\bullet h(e) \setminus h(e)^\bullet)$, and $c \in {}^\bullet e$ and $d \in {}^\bullet f$ be two p-labelled conditions (see Figure 4.2). Since e and f are distinct non-conflicting events, c and d are distinct non-conflicting conditions. Hence, as the net system is safe, c and d are not concurrent, i.e., either $c \prec d$ or $d \prec c$. Since the latter contradicts $f \in (e^\bullet)^\bullet$, we have $c \prec d$. Now, as $p \notin h(e)^\bullet$, there must be a condition $b \in e^\bullet$ such that $h(b) \neq p$ and $b \prec d$, and so e and f are separated. $\square$

Corollary 4.4. *Let π be a branching process of a safe net system, e be one of its events, and (t, D) be a (π, e)-extension. Then ${}^\bullet t \cap ({}^\bullet h(e) \setminus h(e)^\bullet) = \emptyset$.*

In view of the above corollary, the algorithm may consider only the transitions from the set $(u^\bullet)^\bullet \setminus ({}^\bullet u \setminus u^\bullet)^\bullet$ rather than $(u^\bullet)^\bullet$ as the candidates for insertion after a u-labelled event e. The resulting algorithm for updating the set of possible extensions after inserting an event e into the unfolding is shown in Figure 4.3. In order to efficiently find all the conditions which are concurrent to a condition d, one can maintain the concurrency relation, as suggested in [29]. However, such an approach is not suitable if we aim at producing large unfoldings. Another way is to mark in the procedure Cover all the conditions which are not concurrent to d as unusable, and unmark them during the backtracking.

It is interesting to apply this technique in the special case where a transition t has a self-loop, i.e., $t \in (t^\bullet)^\bullet$ (clearly, if $t \notin (t^\bullet)^\bullet$ then a new instance of t cannot be inserted after an event marked by t). We consider the following cases:

- ${}^\bullet t \subset t^\bullet$. Then t must be dead, otherwise the net is not safe (and even unbounded). Indeed, executing t produces all necessary tokens for t to be executed again. Therefore, if t is not dead, it can be executed successively an arbitrary number of times, producing arbitrary many tokens on the places from $t^\bullet \setminus {}^\bullet t \neq \emptyset$.

- ${}^\bullet t = t^\bullet$ (such 'strange' transitions do appear in some of the examples attempted in Section 4.6, e.g., in the Elev(n) series described in Chapter 3). Then, according to Proposition 4.1, for any instance e of t occurring in the built part of the prefix, only one instance f of t (with ${}^\bullet f = e^\bullet$) can be inserted directly after e, producing in the case of a dense cutting context a cut-off event (note that $\mathit{Mark}([e]) = \mathit{Mark}([f])$, and $[e] \subset [f]$, i.e., $e \lhd f$ for any adequate order $\lhd$). Moreover, in the case of a saturated cutting context, every instance e of t is a cut-off event, with the corresponding configuration $[e] \setminus \{e\}$, and so we do not have to insert other events after it.

- ${}^\bullet t \cap t^\bullet \neq \emptyset$ and ${}^\bullet t \not\subseteq t^\bullet$. Then, by Corollary 4.4, another instance of t cannot be inserted directly after t.

As a result, a new instance of t can be inserted after a t-labelled event only if ${}^\bullet t = t^\bullet$.

```
function UpdatePotExt(π, e)
    extensions ← ∅  /* global */
    u ← h(e)
    for all t ∈ (u•)• \ (•u \ u•)• do
        preset ← {b ∈ e• | h(b) ∈ •t} /* not complete yet */
        C ← all conditions of π concurrent to e
        Cover(C, t, preset)
    return extensions

procedure Cover(C, t, preset)
    if |•t| = |preset|
    then extensions ← extensions ∪ {(t, preset)}
    else
        choose p ∈ •t \ h(preset)
        for all d ∈ C such that h(d) = p do
            C′ ← {c ∈ C | c co d}
            Cover(C′, t, preset ∪ {d})
```

Figure 4.3: An algorithm for updating the set of possible extensions.

4.3 Preset trees

The presets of candidate transitions for inserting after an event e have often common parts besides the places from $h(e)^\bullet$, and the algorithm may be finding instances of the same places in the generated part of the prefix for several times. To avoid this, one may identify the common parts of the presets, and treat them only once. The main idea is illustrated by the following example.

Let e be the last event inserted into the prefix being built and $h(e)^\bullet = \{p\}$. Moreover, let t_1, t_2, t_3, and t_4 be the possible candidates for inserting after e such that ${}^\bullet t_1 = \{p, p_1, p_2, p_3, p_4\}$, ${}^\bullet t_2 = \{p, p_1, p_2, p_3\}$, ${}^\bullet t_3 = \{p, p_1, p_2, p_3, p_5\}$, and ${}^\bullet t_4 = \{p, p_2, p_3, p_4, p_5\}$. The condition labelled by p in each case comes from $e^\bullet$. Hence, to insert t_i, the algorithm has to find a co-set C_i such that e *co* C_i and $h(C_i) = {}^\bullet t_i \setminus \{p\}$ (if there are several such co-sets, then several instances of t_i should be inserted). By gluing the common parts of the transition presets, we can obtain a tree shown in Figure 4.4(a), which can then be used to simplify the task of finding the co-sets C_i. Formally, we proceed as follows.

Definition 4.5. Let u be a transition of a net system Σ and $U = (u^\bullet)^\bullet \setminus ({}^\bullet u \setminus u^\bullet)^\bullet$. A *preset tree* of u, PT_u, is a directed tree satisfying the following:

- Each vertex is labelled by a set of places, so that the root is labelled by $\emptyset$, and the sets labelling the nodes of any directed path are pairwise disjoint.
- Each transition $t \in U$ has a unique associated vertex v, such that the union

of all the place sets along the path from the root to v is equal to ${}^\bullet t \setminus u^\bullet$ (different transitions may have the same associated vertex).

- Each leaf is associated to at least one transition (unless the tree consists of one vertex only).

The *weight* of PT_u is defined as the sum of the weights of all the nodes, where the *weight* of a node is the cardinality of the set of places labelling it.

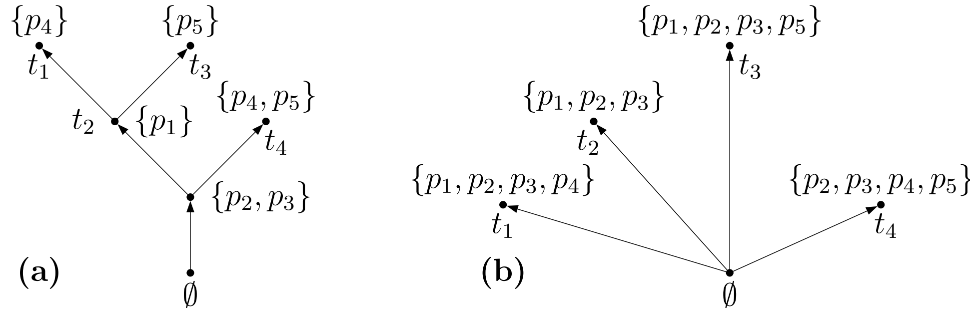

Figure 4.4: An optimized preset tree **(a)** of weight 7, and a non-optimized one **(b)** of weight 15.

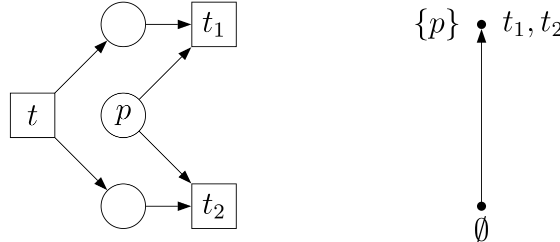

Figure 4.5: Using preset trees may be useful even if $\textsc{PreMax}_\Sigma = 2$ and no two transitions have the same preset.

Having built a preset tree, one can use the algorithm shown in Figure 4.6 to update the set of possible extensions, aiming at avoiding redundant work. As shown in Figure 4.5, there might be gains even when no two transitions have the same preset and $\textsc{PreMax}_\Sigma = 2$, where $\textsc{PreMax}_\Sigma \stackrel{\text{df}}{=} \max_{t\in T} |{}^\bullet t|$ denotes the size of the maximal transition preset in Σ. Note that we only need one preset tree PT_u per transition u of a net system, which can be built during the preprocessing stage.

4.4 Building preset trees

Two problems which we now address are: (i) how to evaluate the 'quality' of preset trees, and (ii) how to efficiently construct them. If we use the 'totally non-optimized' preset tree shown in Figure 4.4(b) instead of that in Figure 4.4(a)

```
function UpdatePotExt(π, e)
    extensions ← ∅  /* global */
    tree ← preset tree for h(e)  /* pre-calculated */
    C ← all conditions of π concurrent to e
    Cover(C, tree, e, ∅)
    return extensions

procedure Cover(C, tree, e, preset)
    for all transitions t labelling the root of tree do
        extensions ← extensions ∪ {(t, preset ∪ {b ∈ e• | h(b) ∈ •t})}
    for all sons tree′ of tree do
        R ← places labelling the root of tree′
        for all co-sets CO ⊆ C such that h(CO) = R do
            Cover({c ∈ C | c co CO}, tree′, e, preset ∪ CO)
```

Figure 4.6: An algorithm for updating the set of possible extensions.

as an input to the algorithm in Figure 4.6, it will work in a way very similar to that of the standard algorithm in Figure 4.3, trying the candidate transitions one-by-one. However, gluing the common parts of the presets decreases both the weight of the preset tree and the number of times the algorithm attempts to find new conditions concurrent to the already constructed part of event presets. This suggests that preset trees with small weight should be preferred. Such a 'minimal weight' criterion may be seen as rather rough, since it is hard to predict during the preprocessing stage which preset tree will be better, as different ones might be better for different instances of the same transition. Another problem is that the reduction of the weight of a preset tree leads to the creation of new vertices and splitting of the sets of places among them, effectively decreasing the weight of a single node. This may affect the efficiency of heuristics which potentially might be used for finding co-sets in the algorithm in Figure 4.6. But in our experiments this drawback was usually more than compensated for by the speedup gained by merging the common parts of the work spent on finding co-sets forming the presets of newly inserted events.

Since there may exist a whole family of minimal-weight preset trees for the same transition, one could improve the criterion by taking into account the remark about heuristics for finding co-sets, and prefer minimal weight preset trees which also have the minimal number of nodes. Furthermore, one could assign coefficients to the vertices, depending on the distance from the root, the cardinality of the labelling sets of places, etc., and devise more complex optimality criteria. However, this may get too complicated and the process of building preset trees can easily become more time consuming then the unfolding itself. And, even if a very complicated criterion is used, the time spent on building a highly optimized

preset tree can be wasted: the transition may be dead, and the corresponding preset tree will never be used by the unfolding algorithm.[4] Therefore, in the actual implementation, we decided to adopt the simple minimal-weight criterion and, in the view of the next result, it was justifiable to implement a relatively fast greedy algorithm aiming at 'acceptably light' preset trees.

Proposition 4.6. *Deciding whether there exists for a given transition t of a net system Σ a preset tree of at most the given weight W is an NP-complete problem in the size of Σ. Moreover, it remains NP-hard even in the case* $\textsc{PreMax}_\Sigma = 3$.

Proof.[5] The problem is in NP as the size of a preset tree is polynomial in the size of Σ, and we can guess it and check its well-formedness and weight in polynomial time.

The proof of NP-hardness is by reduction from the vertex cover problem. Given an undirected graph $G = (V, E)$, construct Σ as follows: take $V \cup \{p', p''\}$, where $p', p'' \notin V \cup E$ are new elements, as the set of places, and for each edge $\{v_1, v_2\} \in V$ take a transition with $\{p'', v_1, v_2\}$ as its preset. Moreover, take another transition t such that ${}^\bullet t = \{p'\}$ and $t^\bullet = \{p''\}$ (note that all the other transitions belong to $(t^\bullet)^\bullet$). There is a one-to-one correspondence between preset trees for t and vertex covers for G, such that the weight of a preset tree is equal to the size of the corresponding vertex cover. Therefore, the problem of deciding whether there is a preset tree of at most the given weight W is NP-hard; moreover, $\textsc{PreMax}_\Sigma = 3$ for our construction, and so the problem is NP-hard even in such a restricted case. □

In Figure 4.7 we outlined simple bottom-up and top-bottom algorithms for solving this problem. Note that the input in each case is a set of sets of places $\{A_1, \dots, A_k\} = \{{}^\bullet u \setminus t^\bullet \mid u \in U\} \cup \{\emptyset\}$. As it is trivial to assign vertices to the transitions, we do not show this part in the algorithms in Figure 4.7. We denote by $Tree(v, \{Tr_1, \dots, Tr_l\})$ a tree with the root v and sons $Tr_1, \dots, Tr_l$, which are trees, and use $\cdot$ instead of the set of son trees if their identities are irrelevant. Moreover, in the further discussion, we will often identify a tree with the set of places at its root, provided that this does not create an ambiguity. We also adopt the standard notation $\bigcap S \stackrel{\text{df}}{=} \bigcap_{A \in S} A$ and $\bigcup S \stackrel{\text{df}}{=} \bigcup_{A \in S} A$.

The bottom-up algorithm first computes the root of the preset tree being built as the intersections of the sets in S, and this intersection is removed from all these sets. Then it chooses a place p belonging to the maximum number of sets, removes it from there, and recursively builds a preset tree for such sets, subsequently adding p to its root. The recursively processed sets are removed from further consideration, and the process is repeated for the remaining sets,

[4] In the $\textsc{Dpfm}(11)$ example (see Chapter 3), the net has 5633 transitions, but the built complete prefix has only 199 non-cut-off events. For this benchmark, the preprocessing stage took more time then the process of unfolding, even though the simple 'minimal weight' criterion was used. This suggests that one might generate preset trees 'on demand' during a run of the unfolding algorithm and cache them.

[5] The main idea of this proof was suggested by Peter Rossmanith.

```
function BuildTree(S = {A_1, ..., A_k})    /* bottom-up */
   root ← ⋂ S
   S ← {A \ root | A ∈ S}
   TS ← ∅
   while ⋃ S ≠ ∅ do  /* while there are non-empty sets */
      choose p ∈ ⋃ S such that |{A ∈ S | p ∈ A}| is maximal
      Tree(v, ts) ← BuildTree({A \ {p} | A ∈ S ∧ p ∈ A})
      TS ← TS ∪ {Tree(v ∪ {p}, ts)}
      S ← {A ∈ S | p ∉ A}
   return Tree(root, TS)

function BuildTree({A_1, ..., A_k})    /* top-down */
   TS ← {Tree(A_1, ∅), ..., Tree(A_k, ∅)}
   while |TS| > 1 do
      choose Tree(A′, ·) ∈ TS and Tree(A″, ·) ∈ TS
            such that A′ ≠ A″ and |A′ ∩ A″| is maximal
      I ← A′ ∩ A″
      T_⊂ ← {Tree(B \ I, ts) | Tree(B, ts) ∈ TS ∧ I ⊂ B}
      TS ← {Tree(B, ·) ∈ TS | I ⊄ B} ∪ {Tree(I, T_⊂)}
   /* |TS| = 1 */
   return the remaining tree Tr ∈ TS
```

Figure 4.7: Two algorithms for building trees.

until there are no more non-empty ones. The resulting preset tree comprises the root computed in the beginning of the algorithm, and the set of son trees built recursively.

In contrast to the bottom-up algorithm, the top-bottom one builds a preset tree starting from the leafs. Each set in the list of parameters is considered as a preset tree consisting of one node, and the set of such trees is stored in TS. On each iteration of the main loop, a set I is computed as an intersection having the maximal cardinality of two distinct sets in TS. Then all the preset trees in TS with roots labelled by supersets of I are extracted from TS and merged into a single preset tree, obtained by subtracting I from all these supersets and taking it as the new root and these trees as the sons. This new tree is added to TS, and the process is continued until only one tree is left in TS; it then is returned as the result.

Note that in this algorithm the value of I cannot be the same on two different iterations of the main loop, because on the first of such two iterations the supersets of I are removed from TS, and I is added there. After this, all the pairwise intersections of the sets which ever appear in TS are not supersets of I. Therefore, if on some iteration, right after computing I, there is a tree $\mathit{Tree}(I, ts) \in TS$,

then it is a leaf, i.e., $ts = \emptyset$. Indeed, otherwise the set I labelling its root would have also been selected on an earlier iteration, a contradiction. Thus only proper supersets of I in TS are considered by the algorithm when it computes the set $T_{\subset}$ of sons of the tree being built on the current iteration.

The described two algorithms do not necessarily give an optimal solution, but they do allow in many cases to produce a significantly 'lighter' tree compared to the totally non-optimized one (see the W_{rat} column in Tables 4.1–4.3). We implemented them both, to check which approach performs better. The tests indicated that in most cases the produced trees had the same weight, but sometimes the bottom-up approach suffered from the effect which can be illustrated by the following example. Let $A_1 = \{p_1, \ldots, p_{10}\}$, $A_2 = \{p_2, \ldots, p_{10}\}$, $A_3 = \{p_1, p_{11}\}$, and $A_4 = \{p_1, p_{12}\}$. On the first iteration of the algorithm p_1 is chosen, and this results in the tree of weight 21, shown in Figure 4.8(a), whereas it is possible to build a tree of weight 13 (Figure 4.8(b)).

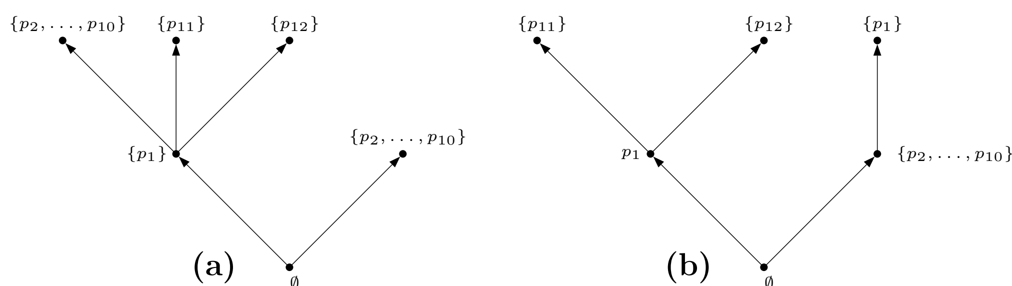

Figure 4.8: A tree of weight 21, produced by the bottom-up algorithm **(a)** and a tree of weight 13, corresponding to the same sets **(b)**.

The top-down algorithm is more stable, and only in rare (see, for example, Figure 4.9) cases it produces 'heavier' trees then the bottom-up one. Therefore, we will concentrate on the top-down algorithm and its efficient implementation.

4.5 Implementation issues

A sketch of a possible implementation of the top-down algorithm for building preset trees is shown in Figure 4.10, and Figure 4.11 illustrates its work. Note that the size of any set which can appear during the calculations does not exceed $\textsc{PreMax}_{\Sigma}$. Therefore, we can assume that the cardinalities are attached to the sets, and the other operations on sets (we only need to compute intersection of sets, cardinality of a set, and check set inclusion and equality) can be performed in $O(\textsc{PreMax}_{\Sigma})$ worst case time. In practice, the sets usually quickly degrade to singletons or to the empty set, so the calculations are quite fast.

The idea of the algorithm is to compute all pairwise intersections of the sets in TS before the main loop starts, and then maintain this data structure. On each

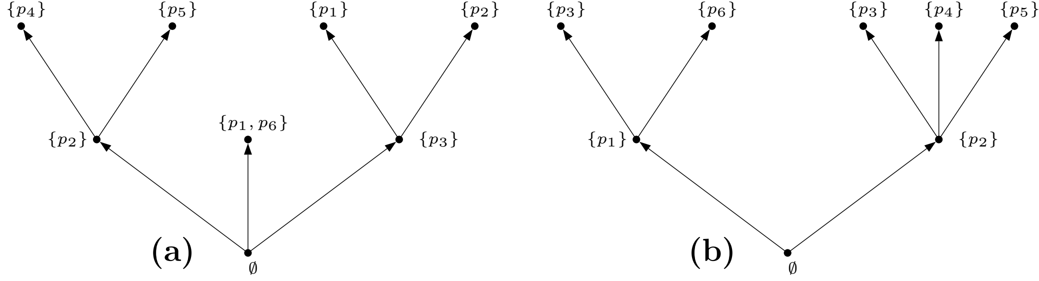

Figure 4.9: A tree of weight 8, produced by the top-down algorithm **(a)** and a tree of weight 7, produced by the bottom-up algorithm **(b)**. Both trees correspond to the sets $A_1 = \{p_1, p_3\}$, $A_2 = \{p_1, p_6\}$, $A_3 = \{p_2, p_3\}$, $A_4 = \{p_2, p_4\}$, and $A_5 = \{p_2, p_5\}$. The intersection $\{p_3\} = A_1 \cap A_3$ was chosen on the first iteration of the top-down algorithm.

step, the algorithm chooses a set I of maximal cardinality from *Intersections*, and updates the variables *TS* and *Intersections* in the following way. It finds all the supersets of I in *TS*, and removes them (this can be done using $O(|TS|)$ operations with sets, if *TS* is implemented as, e.g., a double linked list). Moreover, the algorithm also removes all the intersections from *Intersections* corresponding to these sets. Then the intersections of I with the remaining sets in *TS* are added to *Intersections*, and I is inserted into *TS*.

Since we have k sets in the beginning of the algorithm, the main loop of the algorithm cannot be executed more then $k - 1$ times, because on each step we remove at least 2 sets from the set *TS* and then add one. Therefore, the number of sets which ever appear in *TS* does not exceed $2k - 1$, i.e., the algorithm performs $O(k^2)$ insertions into *Intersections*, $O(k^2)$ removals from there, and $O(k)$ operations of finding a set with maximal cardinality (note that the size of *Intersections* is $O(k^2)$). Using a suitable data structure such as a red-black tree (see, e.g., [20]) for representing *Intersections*, we can perform any of these operations in $O(\text{PREMAX}_\Sigma \cdot \log k^2) = O(\text{PREMAX}_\Sigma \cdot \log k)$ worst case time (note that in this case we have to introduce another operation on sets, viz. checking if $A_i \prec_{tot} A_j$, where $\prec_{tot}$ is an arbitrary total order, refining the size order $A_i \prec_{size} A_j \Leftrightarrow |A_i| < |A_j|$). Therefore, the worst case time of the algorithm in Figure 4.10 is $O(\text{PREMAX}_\Sigma \cdot k^2 \cdot \log k)$.

One can achieve $O(\text{PREMAX}_\Sigma \cdot k^2)$ average time, using instead of a red-black tree the following simple data structure for *Intersections*. For each possible size of a set (as it was already mentioned, these sizes do not exceed PREMAX_Σ), we keep a double linked list of sets having this cardinality (reducing, thus, inserting a set into *Intersections* to adding it into the list corresponding to its cardinality, which requires just $O(1)$ time). In order to facilitate removing an item from *Intersections*, we can maintain a hash table (note, that in this case we need an additional operation with sets, viz. computing the hash code of a set, which may

function BUILDTREE($\{A_1, \ldots, A_k\}$)
  $TS \leftarrow \{\, Tree(A_1, \emptyset), \ldots, Tree(A_k, \emptyset)\}$
  $Intersections \leftarrow \{\!\!\{ A_i \cap A_j \mid 1 \le i < j \le k \}\!\!\}$ /* multiset of sets */
  while $|TS| > 1$ **do**
    choose $I \in Intersections$ **such that** $|I|$ is maximal
    $T_\subset \leftarrow \{\, Tree(B \setminus I, ts) \mid Tree(B, ts) \in TS \wedge I \subset B\}$
    for all $Tree(A, ts) \in TS$ **such that** $I \subseteq A$ **do**
      $TS \leftarrow TS \setminus \{\, Tree(A, ts)\}$
      $Intersections \leftarrow Intersections - \{\!\!\{ A \cap B \mid Tree(B, \cdot) \in TS \}\!\!\}$
    $Intersections \leftarrow Intersections + \{\!\!\{ I \cap B \mid Tree(B, \cdot) \in TS \}\!\!\}$
    $TS \leftarrow TS \cup \{\, Tree(I, T_\subset)\}$
  /* $|TS| = 1$ */
  return the remaining tree $Tr \in TS$

Figure 4.10: A top-down algorithm for building preset trees.

be assumed to take $O(\textsc{PreMax}_\Sigma)$ in the worst case). With this idea, removal of a set can be done in $O(\textsc{PreMax}_\Sigma)$ average time (in order to remove an element, the algorithm has to make a single lookup in the hash table to find it, which leads on average to $O(1)$ calls to the hash function, each costing $O(\textsc{PreMax}_\Sigma)$ time units; the remaining operations take just $O(1)$ time units). Moreover, if we keep track of the current maximal cardinality (since the size of sets added into *Intersections* cannot exceed the size of the last removed from there set, the number of times the cardinality changes does not exceed the number of possible cardinalities) then the total time spent by algorithm on finding sets with maximal cardinality is $O(k + \textsc{PreMax}_\Sigma)$.

It is essential for the correctness of the algorithm that *Intersections* is a multiset rather than a set, and so we have to keep duplicates in this data structure. It is probably better to implement this by keeping a counter for each set inserted into *Intersections*, rather then by keeping several copies of the same set, since the multiplicity of simple sets (e.g., singletons or the empty set) can by very high (note that inserting a set is now slightly more complicated, but one can use the same hash table as for removing, and perform it in $O(\textsc{PreMax}_\Sigma)$ average time). Moreover, if the multiplicities are calculated, we often can reduce the weights of produced trees. The idea is to choose among the sets with maximal cardinality those which have the maximal number of supersets in TS (this would improve the tree in Figure 4.9(a), forcing $\{p_2\}$ to be chosen on the first iteration). Let us show that such sets have the highest multiplicity among the sets with the maximal cardinality. Indeed, each time at the moment this choice is made by the algorithm, the values of TS and *Intersections* are 'synchronized' in the sense that *Intersections* contains all pairwise intersections of the sets from TS, with proper multiplicities. Now, let $I \in Intersections$ be a set with the maximal car-

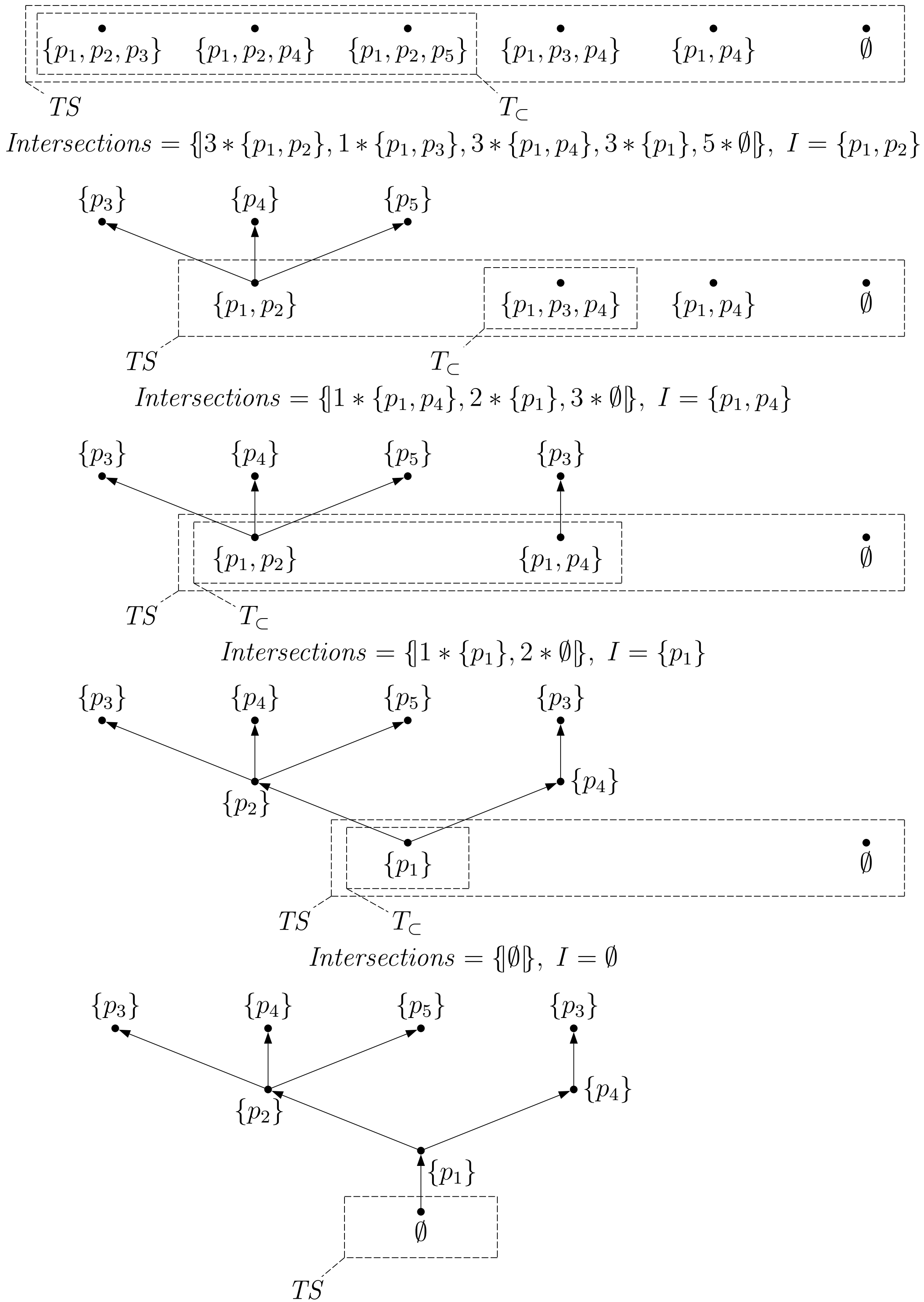

Figure 4.11: An example run of the top-down algorithm shown in Figure 4.10. Snapshots (except the last one) are taken for every iteration of the main loop, before the **for all** loop starts.

dinality, and n be the number of its supersets in TS (note that $n \geq 2$). The intersection of two sets can be equal to I only if they both are supersets of I. Moreover, since there is no set in *Intersections* with cardinality greater then $|I|$, the intersections of any two distinct supersets of I from TS are exactly I. Therefore, the multiplicity of I is $C_n^2 = n(n-1)/2$, for some $n \geq 2$. This function is strictly monotonic for all positive n, i.e., there is monotonic one-to-one correspondence between the multiplicities of sets with the maximal cardinality from *Intersections*, and the numbers of their supersets in TS. Therefore, among the sets of maximal cardinality, those having the maximal multiplicity have the maximal number of supersets in TS.

It is easy to implement this improvement if a red-black tree is used to represent *Intersections*. The only thing to be done is to choose the total ordering on the nodes of the tree refining the following *size-multiplicity* one: $A_i \prec_{sm} A_j \Leftrightarrow |A_i| < |A_j| \vee |A_i| = |A_j| \wedge Intersections(A_i) < Intersections(A_j)$. We may assume that it can be computed in $O(\textsc{PreMax}_\Sigma)$ worst case time, so the worst case complexity of the algorithm remains $O(\textsc{PreMax}_\Sigma \cdot k^2 \cdot \log k)$.

But if we want to use the described above data structure based on a hash table, some changes are needed. The main idea is to 'slice out' the sets from *Intersections* having the maximal cardinality and handle them separately. The remaining sets can be processed in the same way as before (we never have to look for a set with the maximal multiplicity among them, because their cardinalities are not maximal). The multiplicities of the 'sliced out' sets are of the form C_i^2, where $i \leq |TS| \leq k$, and we can hold them in an array of double linked lists, such that the items of the i-th list have the multiplicity C_i^2 (note that the size of the array does not exceed $i_{max} \leq k$, where $C_{i_{max}}^2$ is the maximal multiplicity). This additional data structure can be built in time linear in the number of sets of maximal cardinality in *Intersections*. Therefore, the total time spent on building such a structure is $O(k^2)$, since the total number of sets which appear in *Intersections* is $O(k^2)$.

Now, since the sizes of all the sets which are inserted into *Intersections* are less than the current maximal cardinality, one does not have to modify this additional data structure when inserting new items. Therefore, the only operations we still need are removing a set and finding a set of maximal multiplicity. The latter is simple if we maintain the index of the non-empty list containing the sets of maximal multiplicity. Since we never add new sets to the lists, and the multiplicity of the sets of maximal cardinality can only decrease, the value of this index can only decrease for each particular cardinality (though it may be increased when the algorithm, having exhausted the current 'slice' of sets of the maximal cardinality, switches to smaller ones). Since the sum of i_{max}'s for all slices does not exceed $2k$ (because for each slice at least once we replace $i_{max} \geq 2$ sets from TS by one set), the total number of updates of this index is $O(k + \textsc{PreMax}_\Sigma)$.

Therefore, the only thing which still needs to be explained is how to remove a set of the maximal cardinality from *Intersections*. If the multiplicity of this set is

procedure REMOVE(A)
  if the cardinality of A is not maximal
  then remove A in the usual way
  else
    if the multiplicity of A is 1
    then remove A from the 2nd list /* $1 = C_2^2$ */
    else
      if $\exists i$ **such that** the multiplicity of A is C_i^2
      then
        move A from the i-th list to $(i-1)$-th list
      decrement the multiplicity of A

Figure 4.12: Removing a set from *Intersections*.

one, we can just remove it from the list to which it belongs. But if its multiplicity is greater than one then it is of the form C_i^2, and, since the multiplicities of the sets of the maximal cardinalities corresponds to the number of their supersets in TS, we can be sure that this set will be removed for several more times by the end of the current iteration, so that eventually its multiplicity will be of the form C_j^2 for some $j < i$. Therefore, we can use the algorithm shown in Figure 4.12 (note that we allow the multiplicities of the set in the i-th list to be greater than C_i^2, but it is guaranteed that they will have the proper form by the end of the current iteration of the main loop). The average time for removing an item is $O(\text{PREMAX}_\Sigma)$, thus, this implementation of the algorithm for building trees takes on average $O(\text{PREMAX}_\Sigma{\cdot}k^2)$ time units.

4.6 Experimental results

The results of experiments are summarized in Tables 4.1–4.3, where we use *time* to indicate that the test had not stopped after 15 hours, and *mem* to indicate that the test terminated because of memory overflow. The methodology of testing correctness of the algorithm and the test cases are described in Chapter 3. For comparison, the ERVUNFOLD 4.5.1 tool, available from the Internet, was used. The methods implemented in it are described in [29–31, 85]; in particular, it maintains the concurrency relation.

The meaning of the columns in the tables is as follows (from left to right): the name of the problem; the number of places and transitions and the average/maximal size of transition presets in the original net system; the number of conditions, events and cut-off events in the complete prefix; the time spent by the ERVUNFOLD tool (in seconds); the time spent by the new algorithm on building the preset trees and unfolding the net; the ratio $\text{W}_{rat} = \text{W}_{opt}/\text{W}$, where W_{opt} is the sum of the weights of the constructed preset trees, and W is the sum

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$a/m\|^{\bullet}t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-tr*	*Unf*	W_{rat}
Bds	53	59	1.88/2	12310	6330	3701	1.30	<0.01	3.87	0.53
Ftp(1)	176	529	1.98/2	178085	89046	35197	*time*	0.16	2625	0.52
Q	163	194	1.89/2	16123	8417	1188	8.69	0.03	39.43	0.81
Dpd(4)	36	36	1.83/2	594	296	81	0.01	<0.01	0.02	0.71
Dpd(5)	45	45	1.82/2	1582	790	211	0.04	<0.01	0.16	0.71
Dpd(6)	54	54	1.81/2	3786	1892	499	0.22	<0.01	0.83	0.71
Dpd(7)	63	63	1.81/2	8630	4314	1129	1.16	<0.01	5.49	0.71
Dpfm(2)	7	5	1.80/2	12	5	2	0.00	<0.01	<0.01	1.00
Dpfm(5)	27	41	1.98/2	67	31	20	0.00	0.01	<0.01	1.00
Dpfm(8)	87	321	2/2	426	209	162	0.01	0.08	0.01	1.00
Dpfm(11)	1047	5633	2/2	2433	1211	1012	0.05	89.35	0.74	1.00
Dph(4)	39	46	1.96/2	680	336	117	0.01	<0.01	0.03	1.00
Dph(5)	48	67	1.97/2	2712	1351	547	0.10	<0.01	0.36	1.00
Dph(6)	57	92	1.98/2	14590	7289	3407	2.16	<0.01	9.74	1.00
Dph(7)	66	121	1.98/2	74558	37272	19207	57.43	0.01	263	1.00
Elev(1)	63	99	1.89/2	296	157	59	0.01	0.01	0.01	0.62
Elev(2)	146	299	1.95/2	1562	827	331	0.02	0.13	0.14	0.60
Elev(3)	327	783	1.97/2	7398	3895	1629	0.61	1.59	2.73	0.60
Elev(4)	736	1939	1.99/2	32354	16935	7337	16.15	25.57	68.43	0.61
Furn(1)	27	37	1.65/2	535	326	189	0.01	<0.01	0.02	0.50
Furn(2)	40	65	1.71/2	4573	2767	1750	0.19	<0.01	0.54	0.44
Furn(3)	53	99	1.75/2	30820	18563	12207	8.18	<0.01	29.10	0.41
Gasnq(2)	71	85	1.94/2	338	169	46	0.01	0.01	0.01	0.94
Gasnq(3)	143	223	1.97/2	2409	1205	401	0.09	0.03	0.36	0.96
Gasnq(4)	258	465	1.98/2	15928	7965	2876	4.54	0.10	18.45	0.97
Gasnq(5)	428	841	1.99/2	100527	50265	18751	785	0.32	817	0.98
Gasq(1)	28	21	1.86/2	43	21	4	<0.01	<0.01	<0.01	0.86
Gasq(2)	78	97	1.95/2	346	173	54	<0.01	<0.01	0.02	0.93
Gasq(3)	284	475	1.99/2	2593	1297	490	0.11	0.12	0.40	0.97
Gasq(4)	1428	2705	2/2	19864	9933	4060	7.93	7.91	29.70	0.99
Key(2)	94	92	1.97/2	1310	653	199	0.06	0.01	0.15	0.93
Key(3)	129	133	1.98/2	13941	6968	2911	2.51	0.03	10.48	0.94
Key(4)	164	174	1.98/2	135914	67954	32049	6247	0.06	864	0.94
Mmgt(1)	50	58	1.95/2	118	58	20	0.01	0.01	<0.01	0.66
Mmgt(2)	86	114	1.95/2	1280	645	260	0.03	0.03	0.08	0.64
Mmgt(3)	122	172	1.95/2	11575	5841	2529	1.75	0.07	6.09	0.64
Mmgt(4)	158	232	1.95/2	92940	46902	20957	188	0.14	504	0.64
Over(2)	33	32	1.88/2	83	41	10	<0.01	<0.01	<0.01	0.89
Over(3)	52	53	1.89/2	369	187	53	0.01	0.01	0.01	0.88
Over(4)	71	74	1.89/2	1536	783	237	0.06	<0.01	0.15	0.87
Over(5)	90	95	1.89/2	7266	3697	1232	0.94	<0.01	3.35	0.86
Rw(6)	33	85	1.99/2	806	397	327	0.01	0.01	0.01	1.00
Rw(9)	48	181	1.99/2	9272	4627	4106	0.21	0.03	0.34	1.00
Rw(12)	63	313	2/2	98378	49177	45069	14.46	0.10	15.30	1.00

Table 4.1: Experimental results for standard benchmarks with small transition presets.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	$a/m\|^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-tr*	*Unf*	W_{rat}
Byz	504	409	3.33/30	42276	14724	752	126	0.14	231	0.71
Dme(2)	135	98	3.24/5	487	122	4	0.01	0.01	0.02	0.93
Dme(3)	202	147	3.24/5	1210	321	9	0.07	0.01	0.09	0.93
Dme(4)	269	196	3.24/5	2381	652	16	0.25	0.03	0.34	0.93
Dme(5)	336	245	3.24/5	4096	1145	25	0.79	0.03	1.01	0.93
Dme(6)	403	294	3.24/5	6451	1830	36	2.37	0.05	2.96	0.93
Dme(7)	470	343	3.24/5	9542	2737	49	6.37	0.19	7.28	0.93
Dme(8)	537	392	3.24/5	13465	3896	64	14.12	0.09	16.08	0.92
Dme(9)	604	441	3.24/5	18316	5337	81	27.78	0.11	31.82	0.92
Dme(10)	671	490	3.24/5	24191	7090	100	51.67	0.13	58.14	0.92
Dme(11)	738	539	3.24/5	31186	9185	121	89.18	0.16	98.96	0.92
Sync(2)	72	88	1.89/3	3884	2091	474	0.29	<0.01	1.38	0.91
Sync(3)	106	270	2.21/4	28138	15401	5210	14.15	0.06	74.84	0.77

Table 4.2: Experimental results for standard benchmarks with larger transition presets.

of the weights of the 'non-optimized' preset trees (see Figure 4.4). This ratio may be used as a rough approximation of the effect of employing preset trees: $W_{rat} = 1$ means that there is no optimization. Note that when transition presets are large, employing preset trees gives certain gains, even if this ratio is close to 1 (see, e.g., the Dme(n) series).

Test cases

We attempted (Tables 4.1–4.3) the set of benchmark examples described in Chapter 3 (some of the examples from this set are not shown in the tables, since they were trivial for both algorithms).

The sizes of transition presets in the examples shown in Table 4.1 do not exceed 2 for all the examples. Thus, the advantage of using preset trees is not very big, and ERVunfold is usually quicker due to maintaining concurrency relation.[6] But when the size of this relation becomes greater then the amount of the available memory, ERVunfold slows down because of page swapping (e.g., in Gasnq(5), Ftp(1), and Key(4) examples). As for the proposed algorithm, though it is usually slower on these examples, its running time on all the examples in Table 4.1 is quite acceptable, and it was superior on the largest ones (Ftp(1) and Key(4)). Moreover, it scales better on some of these benchmarks (e.g., on the Elev(n) and Mmgt(n) series).

In Table 4.2, the results concerning nets with slightly bigger transition presets are shown. The new algorithm, though it does not use concurrency relation, is almost as fast as ERVunfold, and scales better. The only example in this set with big maximal preset is Byz, but it in fact has only one transition with the preset of size 30, and one transition with the preset of size 13; the sizes of the

[6] After we had implemented concurrency relation, the algorithm became much faster, but for these experiments we decided to switch it off in order not to hinder the effect of using preset trees. See also the experiments in Chapter 7, where the concurrency relation was employed.

presets of the other transitions in this net do not exceed 5.

The experimental results shown in the first part of Table 4.3 indicate that on random nets with large presets (the RND(m, n, k) series described in Chapter 3) the new algorithm significantly outperforms ERVUNFOLD.

One can argue that random nets is not a practical example, and that in practice the presets of transitions are small (and, in fact, do not contain more than two places in the case of binary synchronization). So we tried to spot areas where nets with large transition presets naturally arise. The first candidate is data intensive applications, where processes being modelled compute functions depending on many variables. As an example, we modelled priority arbiters based on dual-rail logic (see Chapter 3). The results are summarized in the second part of Table 4.3. The new algorithm scales better and was able to produce much larger unfoldings.

We expect that many other areas where Petri nets with large presets are needed will be found. Potentially, such nets appear as the result of net transformation, e.g., introducing complementary places or converting bounded nets into safe ones.[7] But even for nets with small transition presets the new algorithm is quite quick (the only reason it was slower than ERVUNFOLD for the examples in Tables 4.1 and 4.2 is because we switched off the concurrency relation).

4.7 Conclusions

In this chapter we have proposed an efficient method of generating possible extensions of a branching process, which is the most time-consuming part of unfolding algorithms. Experimental results indicate that the proposed algorithm can build quite large unfoldings in reasonable time, and have significant advantages for nets with large transition presets.

[7]Such transformation is the basis of the unfolding algorithm for arbitrary bounded nets described in [31].

Problem	Net			Unfolding			Time, [s]			
	$\lvert S\rvert$	$\lvert T\rvert$	$a/m\lvert^\bullet t\rvert$	$\lvert B\rvert$	$\lvert E\rvert$	$\lvert E_{cut}\rvert$	*ERV*	*p-tr*	*Unf*	W_{rat}
RND(5,4,500)	20	520	4.85/5	20814	5645	4712	1.86	8.02	0.79	0.30
RND(5,5,500)	25	525	4.81/5	55698	14029	11689	11.45	7.36	3.66	0.39
RND(5,6,500)	30	530	4.77/5	84451	21774	17269	31.43	8.68	12.21	0.44
RND(5,7,500)	35	535	4.74/5	144700	36019	28922	82.92	8.90	30.69	0.50
RND(5,8,500)	40	540	4.70/5	235600	56691	46559	196	8.79	62.96	0.54
RND(5,9,500)	45	545	4.67/5	304656	72895	59840	324	7.43	105	0.58
RND(5,10,500)	50	550	4.64/5	419946	98477	82279	554	9.07	160	0.60
RND(5,11,500)	55	555	4.60/5	573697	132344	112310	994	6.20	246	0.63
RND(5,12,500)	60	560	4.57/5	627303	145378	122465	1187	5.72	322	0.65
RND(5,13,500)	65	565	4.54/5	718762	166093	140147	1560	5.27	420	0.67
RND(5,14,500)	70	570	4.51/5	802907	185094	156417	1952	5.58	507	0.69
RND(5,15,500)	75	575	4.48/5	842181	195228	163722	6685	6.63	616	0.70
RND(5,16,500)	80	580	4.45/5	886158	206265	171957	*time*	7.10	717	0.71
RND(5,17,500)	85	585	4.42/5	987605	229284	191576	—	3.78	863	0.72
RND(5,18,500)	90	590	4.39/5	1025166	239069	198524	—	5.62	998	0.73
RND(10,2,500)	20	520	9.65/10	34884	7136	6125	12.46	7.34	1.14	0.25
RND(10,3,500)	30	530	9.49/10	1415681	153628	144548	1638	3.90	82	0.49
RND(10,4,500)	40	540	9.33/10	2344821	252320	237000	*mem*	3.51	207	0.59
RND(10,5,500)	50	550	9.18/10	2485903	271083	250600	—	7.90	331	0.64
RND(10,6,500)	60	560	9.04/10	2535070	280560	255010	—	11.32	485	0.67
RND(10,7,500)	70	570	8.89/10	2537646	285323	254767	—	11.91	663	0.70
RND(10,8,500)	80	580	8.76/10	2534970	289550	254000	—	14.84	872	0.72
RND(15,2,500)	30	530	14.21/15	1836868	135307	128358	*mem*	32.28	70.24	0.37
RND(15,3,500)	45	545	13.84/15	3750719	271074	255560	—	14.69	259	0.57
RND(15,4,500)	60	560	13.50/15	3787575	280560	257515	—	7.54	456	0.67
RND(15,5,500)	75	575	13.17/15	3795090	288075	257515	—	6.38	718	0.73
RND(20,2,500)	40	540	18.59/20	4744587	256197	245750	*mem*	46.71	176	0.43
RND(20,3,500)	60	560	17.96/20	5040080	280560	260020	—	16.36	427	0.61
RND(20,4,500)	80	580	17.38/20	5050100	290580	260020	—	9.03	771	0.71
SPA(2)	52	37	2.16/4	111	52	4	<0.01	<0.01	<0.01	0.87
SPA(3)	75	57	2.40/4	324	141	19	0.01	0.01	0.01	0.79
SPA(4)	98	81	2.77/5	1048	421	96	0.04	0.01	0.07	0.72
SPA(5)	121	113	3.34/6	3594	1362	457	0.26	0.03	0.53	0.63
SPA(6)	144	161	4.20/7	13334	4860	2145	3.79	0.08	5.51	0.56
SPA(7)	167	241	5.38/8	52516	18712	9937	64.22	0.28	75.54	0.49
SPA(8)	190	385	6.82/9	216772	76181	45774	*time*	1.26	943	0.43
SPA(9)	213	657	8.35/10	920270	320582	209449	—	6.66	12571	0.38
SPA(2,1)	52	37	2.16/4	111	52	4	<0.01	<0.01	<0.01	0.87
SPA(2,2)	98	81	2.77/5	1206	476	110	0.04	0.01	0.10	0.72
SPA(2,3)	144	161	4.20/7	15690	5682	2512	5.53	0.08	8.28	0.56
SPA(2,4)	190	385	6.82/9	253219	88944	52826	*time*	1.29	1326	0.43
SPA(3,1)	75	57	2.40/4	324	141	19	0.01	<0.01	0.02	0.79
SPA(3,2)	144	161	4.20/7	15690	5682	2512	5.49	0.08	9.09	0.56
SPA(3,3)	213	657	8.35/10	1142214	398850	256600	*time*	6.67	20594	0.38
SPA(4,1)	98	81	2.77/5	1048	421	96	0.04	0.01	0.09	0.72
SPA(4,2)	190	385	6.82/9	253219	88944	52826	*time*	1.27	1326	0.43

Table 4.3: Experimental results for the RND(m,n,k), SPA(n), and SPA(m,n) series.

Chapter 5

Parallel Unfolding Algorithm

In view of recent very fast model checking algorithms employing unfoldings (e.g., the integer programming algorithm described in Chapter 7 and the one from [40–42, 44] based on computing a stable model of a logic program), the problem of efficiently building them becomes increasingly important. Chapter 4 addressed this issue, but we feel that generating net unfoldings deserves further investigation.

In this chapter, a parallel unfolding algorithm is described. Besides offering a high degree of parallelism, it has several other important advantages. In particular, if the cutting context is dense and the used adequate order $\lhd$ is total and refines $\lhd_m$, the total number of times two configurations are compared w.r.t. $\lhd$ is reduced down to the number of cut-off events in the resulting prefix. This allows to gain certain speedup even in a sequential implementation. Some other optimizations are also described.

This chapter is based on the results developed in [45, 46, 60, 61].

5.1 The slicing algorithm

We will use the slicing algorithm shown in Figure 2.6 as the basis for subsequent parallelization. As explained in Chapter 4, the slowest part of unfolding algorithms is computing the set of possible extensions carried out on each iteration of the main loop of the algorithm (a decision version of this problem is, in fact, NP-complete, see [32,42]), and in this chapter we will concentrate on distributing this task among several processors.

Similarly as it was done for the basic algorithm in Chapter 4, one can replace the call $\textsc{PotExt}(\mathit{Pref})$ in the body of the main loop of the slicing algorithm by a call $\textsc{UpdatePotExt}(\mathit{Pref}, e)$, which finds all the possible (Pref, e)-extensions. The resulting algorithm is shown in Figure 5.1. Note that the reason for using the slicing version of the unfolding algorithm rather than the basic one for parallelization is that the iterations in its internal loop are *almost* independent and can be executed in parallel. Of course, for such a scheme to work, one has to provide a way of choosing a slice satisfying the definition of a slice formulated

input : $\Sigma = (N, M_0)$ — a net system
output : *Pref* — the canonical prefix of Σ's unfolding (if it is finite)

```
Pref ← the empty branching process
pe ← {⊥}
cut_off ← ∅
while pe ≠ ∅ do
   choose Sl ∈ SLICES(pe)
   pe ← pe \ Sl
   for all e ∈ Sl in any order refining ◁ do
      if e is a cut-off event of Pref
      then cut_off ← cut_off ∪ {e}
      else
         Pref ← Pref ⊕ {e}
         pe ← pe ∪ UPDATEPOTEXT(Pref, e)
Pref ← Pref ⊕ cut_off
```

Note: e is a cut-off event of *Pref* if there is $C \in \mathcal{C}_e$ such that
the events of C belong to *Pref* but not to *cut_off*, $C \approx [e]$, and $C \lhd [e]$.

Figure 5.1: An improved version of the slicing unfolding algorithm.

in Chapter 2[1] and remove the remaining dependencies between the iterations of the internal loop.

5.2 Choosing a slice

When $\lhd$ refines $\lhd_m$ (in particular, this is the case for all the adequate orders described in Chapter 2), there is a simple scheme for choosing an appropriate set $\textsc{Slices}(pe)$, by setting it to contain all non-empty closed w.r.t. $\lhd$ sets of events from pe whose local configurations have the minimal size. Such a set satisfies the definition of a slice formulated in Chapter 2. Indeed, suppose that $e \in Sl \in \textsc{Slices}(pe)$ and g be an event of Unf_Σ^{max}. If $f \prec g$ for some $f \in pe$ then it is the case that $|[g]| > |[e]|$. Hence, since $\lhd$ refines $\lhd_m$, $g \not\lhd e$. Moreover, if $g \in pe \setminus Sl$ then $g \not\lhd e$ as Sl is a closed w.r.t. $\lhd$ set of events from pe.

Notice that in order to achieve better parallelization, it is advantageous to choose large slices. Therefore, one can simply choose as a slice the set of *all* events from pe, whose size of the local configuration is minimal (note that this set is closed w.r.t. $\lhd$, and, therefore, is in $\textsc{Slices}(pe)$). With this scheme, we

[1]Note that the definition of a slice contains a quantification over the events of the full unfolding, and thus cannot be directly checked.

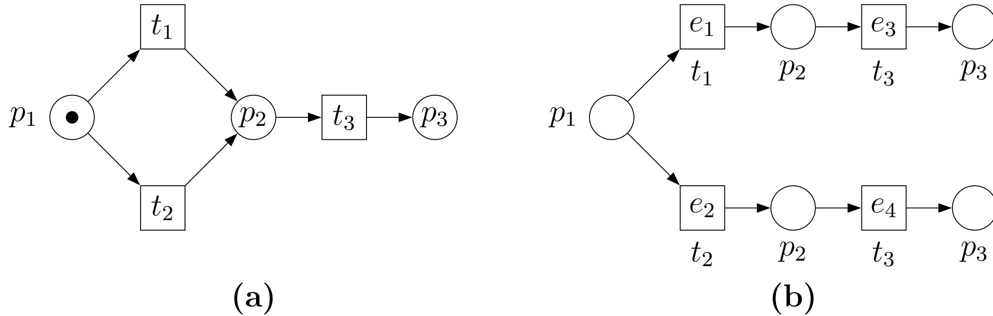

Figure 5.2: A Petri net **(a)** and its unfolding **(b)**. Assuming the cutting context Θ_{ERV}, e_2 is a cut-off event. However, the naïvely parallelized slicing algorithm is not always able to detect this when it processes the slice $\{e_1, e_2\}$, since parallel tasks can be scheduled in such a way that e_2 is processed before e_1. This results in erroneously adding e_4 to the prefix being built.

may simply consider *pe* as a sequence $Sl_1, Sl_2, \ldots$ of sets of events such that Sl_i contains the events whose local configurations have the size i (clearly, in each step of the algorithm there is only a finite number of non-empty Sl_i's). Thus inserting an event e into the queue is reduced to adding it into the set $Sl_{|[e]|}$, and choosing a slice in the main loop of the algorithm can be replaced by a call to $Front(pe)$, returning the first non-empty set Sl_i in *pe*. Now all the required operations with the queue can be performed without comparisons of configurations at all.

5.3 Cut-offs 'in advance'

The main dependency between the iterations of the internal loop is that they must be executed in an order consistent with $\lhd$. It is essential for correct parallelization to remove this dependency (Figure 5.2 shows that one cannot parallelize the **for all** loop of the slicing algorithm by simply ignoring this dependency). One can see that the consistency with $\lhd$ is required only for detecting cut-off events properly, but the order of computing the possible extensions does not matter. Therefore, if the cut-off events appearing in the slice were identified and removed from it 'in advance', the restriction that the events from *Sl* must be processed in an order consistent with $\lhd$ can be safely left out. In the rest of this section we describe a new policy for computing cut-off events, so that when a new slice is chosen by the algorithm it is guaranteed that it does not contain them.

This can be achieved by checking the cut-off criterion each time a new possible extension e is computed rather than in the main loop of the algorithm. Such a strategy introduces several complications. In particular, one has to look for the

corresponding configurations in $\mathit{Pref} \oplus \mathit{pe}$ rather than in *Pref*. Moreover, the cut-off criterion should be checked not only for e, but for all events in *pe* (since e might be a part of a corresponding configuration of some event in the queue). But this strategy has also clear advantages: besides making the iterations of the internal loop independent, it allows one to move the code computing the cut-off criterion into the part of the algorithm which is executed in parallel. Surprisingly, it even allows for significant reduction of the number of times the order $\lhd$ is computed, gaining speedup compared to the traditional strategy (see Section 5.5).

5.4 Parallelizing the unfolding algorithm

Taking into account the ideas from the previous section, the events in *Sl* can be processed in any order. This leads to a possibility of parallelizing the unfolding algorithm when $|\mathit{Sl}| > 1$. But still there is one more problem: the (Pref, f)-extensions for $f \in \mathit{Sl}$ may have in their presets conditions produced by other events from *Sl*, inserted into the prefix before f. This can be achieved by inserting all the events from *Sl* into *Pref* before the loop for computing possible extensions starts, and ignoring some of the inserted events in UpdatePotExt in order to prevent computation of duplicates for some of the possible extensions.

The resulting algorithm is shown in Figure 5.3. Note that the possible extensions are now computed in parallel, and the calls to UpdatePotExt are not interlocked. The critical section is still quite long, but it can be implemented very efficiently and with minimum interlocking (see Section 5.5).

Since UpdatePotExt is the most time-consuming part of the algorithm, this strategy usually provides quite a good parallelization. In most of the performed experiments, there were less then 200 iterations of the main loop, so the time spent on executing the sequential parts of the algorithm was negligible (this fact was confirmed by profiling the program). The first and the last few iterations usually allowed to execute 5–20 ProcessEvent's in parallel (which is already enough to provide quite good parallelism for most existing shared memory architectures), whereas the middle ones were highly parallel (from several hundreds up to several thousands tasks could potentially be executed in parallel). Thus the scalability of the algorithm is usually very good.

Of course, bad examples do exist, in particular those having 'long and narrow' unfoldings, e.g., the Buf(100) net described in Chapter 3 (see the experiments in Section 5.6). But such examples are rare in practice. Intuitively, they have only a small number of different partial order executions of the same length. This means that they have a very small number of conflicts and a low degree of concurrency (as for the Buf(100) example, it has no conflicts at all and allows only few transitions to be executed concurrently). Experiments show that as soon as first conflicts are encountered and added into the prefix being built, the number of events in $\mathit{Front}(\mathit{pe})$ grows rapidly from step to step.

```
input : Σ = (N, M_0) — a bounded net system
output : Pref — a finite and complete prefix of Σ's unfolding

Pref ← the empty branching process
pe ← {⊥}
cut_off ← ∅
while pe ≠ ∅ do
   Sl ← Front(pe)
   pe ← pe \ Sl
   Pref ← Pref ⊕ Sl /* events are numbered when added to Pref */
   for all e ∈ Sl do parallel
      ProcessEvent(e)
Pref ← Pref ⊕ cut_off

procedure ProcessEvent(e)
Pref_⌈e⌉ ← the prefix induced by the events with
      numbers not greater than that of e
for all g ∈ UpdatePotExt(Pref_⌈e⌉, e) do
   begin critical section
      pe ← pe ∪ {g}
      Compute the set S ⊆ pe of cut-off events of Pref ⊕ pe
      cut_off ← cut_off ∪ S
      pe ← pe \ S
   end critical section
```

Note: e is a cut-off event of *Pref* if there is $C \in \mathcal{C}_e$ such that the events of C belong to *Pref* but not to *cut_off*, $C \approx [e]$, and $C \lhd [e]$.

Figure 5.3: A parallel algorithm for unfolding Petri nets.

We implemented the proposed algorithm on the shared memory architecture. It should not be hard to implement it on the distributed memory or even hybrid architecture, consisting of a network of multiprocessors. In that case, each node keeps a local copy of the built part of the prefix and synchronizes it with the master node at the beginning of each iteration of the main loop. The master node is responsible for maintaining the queue of possible extensions, checking the cut-off criterion, and for distributing the work between the slaves; the slaves compute possible extensions and send them to the master. Such a scheme guarantees that the amount of message passing is linear in the size of the resulting prefix.

5.5 Implementation issues

In this section, we concentrate on efficient implementation of the parallel unfolding algorithm. In particular, we suggest how to avoid unnecessary interlocking. We assume that the cutting context is such that $\lhd$ refines $\lhd_m$ and $\mathcal{C}_e = \mathcal{C}_{loc}$ for all $e \in E$.

5.5.1 Minimizing interlocking

It is a well-known fact that in order to gain efficiency one should minimize interlocking and make critical sections of parallel algorithms as 'short' as possible. This can be achieved by making computations as local as possible. For example, the algorithm in Figure 5.3 has to guard the access to the variable *cut_off*. By distributing this variable between the working threads one can avoid contention when accessing it, and the final value of this variable can be composed as the union of the corresponding local values.

Similarly, the queue *pe* can also be distributed. Each thread can have its own local queue to avoid interlocking when inserting a new element into it. In this case, the function *Front* has to compose the slice of several parts coming from different local queues. But since the slice is calculated in the sequential part of the algorithm, interlocking is not needed.

An efficient implementation of the cut-off criterion check is of paramount importance and thus should be discussed in more detail. Due to the restrictions assumed in the beginning of this section, an event e is a cut-off event of *Pref* iff there is an event f in $\mathit{Pref} \oplus \mathit{pe}$ such that $[f] \approx [e]$ and $[f] \lhd [e]$. In the case of $\approx_{mar}$, [30, 31, 85] suggest to maintain a hash table of final markings of local configurations in the prefix to facilitate the search for corresponding configurations. This can be generalized to arbitrary equivalence relations $\approx$, if one uses equivalence classes of $\approx$ (corresponding to the final markings in the case of $\approx_{mar}$) on local configurations as keys to the hash table.

Suppose e is a possible extension computed by the algorithm. Using the hash table it is easy to find all the events $f_1, \ldots, f_k$ of $\mathit{Pref} \oplus \mathit{pe}$ such that $[f_1] \approx \cdots \approx [f_k] \approx [e]$; only such events can become cut-off events with $[e]$

as their correspondent configuration, and only the local configuration of such an event can become a corresponding configuration of e, if e is a cut-off event. Therefore, if there is $i \in \{1 \ldots k\}$ such that $f_i \lhd e$ then e is a cut-off event. Otherwise, e is inserted into the queue, and, for every $i \in \{1 \ldots k\}$ such that $e \lhd f_i$, f_i is declared a cut-off event and removed from the queue (f_i cannot be in *Pref* since, by definition, the slice to which it belongs contains e and thus has not been inserted into the prefix yet). Note that the case $f_i \lhd e \lhd f_j$ for some $i, j \in \{1 \ldots k\}$ is not possible, since f_j is a cut-off event with $[f_i]$ as a corresponding configuration, and thus would have been removed from the queue.

One can observe that with such a strategy events whose local configurations are in different equivalence classes of $\approx$ can be processed in parallel, without interlocking. Therefore, it is sufficient to lock only the entries of the hash table corresponding to the hash code of the equivalence class of $\approx$ on $[e]$ rather than the whole hash table. With such a strategy, the contention occurs only when several threads at the same time try to access entries of the hash table having the same hash code, which happens rarely in practice (this fact was confirmed by experiments).

When $\lhd$ is a total adequate order, each time two configurations are compared w.r.t. $\lhd$, one of the events becomes a cut-off event. Thus the number of the performed comparisons is exactly $|E_{cut}|$ (rather than $\Theta(|E| \log |E|)$ as in the basic algorithm),[2] and the algorithm achieves noticeable speedup even when only one processor is available (see Section 5.6). One can reduce the number of comparisons even further, using the fact that the local configurations of the events which are already in the prefix are always less than those of newly computed possible extensions. But this would provide almost no speedup, since in this case the sizes of local configurations to be compared always differ, and so the comparisons are fast (we may assume that the size of the local configuration is attached to an event).

5.5.2 Computing final markings

Often in order to compute the equivalence class of a configuration, one has to compute its final marking. It is possible to use the definition $Mark(C) \stackrel{\mathrm{df}}{=} h\{\!|Cut(C)|\!\}$, where $Cut(C) \stackrel{\mathrm{df}}{=} \left(\bigcup_{e \in C} e^\bullet\right) \setminus \left(\bigcup_{e \in C} {}^\bullet e\right)$ (note that $\bot \in C$), but this approach is not efficient.

[2]In the basic algorithm, $|E|$ events in total pass through the queue, i.e., there are $|E|$ insertions and $|E|$ extractions of the minimal (w.r.t. $\lhd$) event from pe. Assuming that pe is implemented as a binary heap (and thus any of these two operations can be performed using $\Theta(\log |pe|)$ comparisons), and that the maximal size of pe is $\Theta(|E|)$ (which is indeed the case for 'wide' prefixes), the basic algorithm performs $\Theta(|E| \log |E|)$ comparisons in total. The new algorithm does not extract minimal (w.r.t. $\lhd$) events from pe, and instead computes $Front(pe)$. Thus the binary heap can be replaced by a data structure allowing to perform this operation and an insertion of an event without any comparisons at all (note that $|E_{cut}|$ comparisons are performed by the new algorithm when checking the cut-off criterion).

A better method is to compute the vector $(h\{\!|C|\!\}(t_1), \ldots, h\{\!|C|\!\}(t_n))$, where $t_1, \ldots, t_n$ are the transitions of Σ, and consider it as the Parikh vector x_σ of some linearization σ of the partial order execution represented by C. It can then be used in the marking equation $M = M_0 + \mathfrak{I}_\Sigma \cdot x_\sigma$ (see Section 1.4) to calculate the final marking of C. The advantage of this approach is that the calculation is performed using the original net system rather than the built part of the prefix which can be much larger.

In the case of safe net systems, this approach can be further refined. Indeed, we can perform all calculations modulo 2, which is useful if final markings are represented as bit vectors.

Note that it is simpler to calculate the *marking change vector* $\widetilde{M} = M - M_0$ rather than marking M using the formula $\widetilde{M} = \mathfrak{I}_\Sigma \cdot x_\sigma$. Since $\widetilde{M}$ unambiguously determines M, it can be used as a representation of the final marking of a configuration.

As experiments showed, these simplifications lead to so fast a routine that one can afford recomputing final markings each time they are needed rather than attach them to the corresponding events (we estimate that the time overhead is less than 5% for large prefixes, yet the memory gains may be significant). Note that in such a case searching an entry in a hash table takes more time, since there can be several keys with the same hash code, and each comparison of keys involves computing the final marking of a local configuration. Therefore it makes sense to sort the keys having the same hash code according to some (arbitrary) fixed order, and use a binary rather than linear search when resolving a collision.

5.5.3 Optimizing computation of possible extensions

Since the unfolding algorithm adds events to the prefix being built slice by slice rather than individually, the process of computing possible extensions can be optimized due to merging common parts of the work in the spirit of the preset trees construction described in Chapter 4. This direction is still to be investigated; here we present a simple improvement taking advantage of this idea.

A *cluster* is a non-empty set Cl of $(Pref, e)$-extensions for some event e in $Pref$ such that $Cl \subseteq Sl$. If CP is a partitioning of Sl into non-overlapping clusters then the problem of computing the $(Pref \oplus Sl, e)$-extensions for all $e \in Sl$ can be decomposed in the following way:

$$\bigcup_{e \in Sl} \textsc{UpdatePotExt}(Pref \oplus Sl, e) = \bigcup_{Cl \in CP} \bigcup_{e \in Cl} \textsc{UpdatePotExt}(Pref \oplus Sl, e)$$

Let Cl be a cluster whose elements are $(Pref, e)$-extensions for some event e, and $f \in Cl$. In order to find $(Pref \oplus Sl, f)$-extensions, the algorithm considers the conditions of $Pref \oplus Sl$ which are concurrent to f (only such conditions, together with those from $f^\bullet$, can be in the presets of $(Pref \oplus Sl, f)$-extensions). This can be done by marking the conditions which are f's causal predecessors, or are in conflict with f, as unusable (note that $f \in Sl$ is a maximal (w.r.t. $\prec$)

event of $Pref \oplus Sl$, so the only nodes which are the causal successors of f are the conditions in $f^\bullet$).

Since $e \prec f$ for every event $f \in Cl$, the following holds for any condition b of $Pref \oplus Sl$:

- If $b \prec e$ then $b \prec f$.
- If $b \# e$ then $b \# f$.

In other words, the set of conditions, which are either the causal predecessors of e or in conflict with it, is a common subset of unusable conditions for all $f \in Cl$. Therefore, one can merge the common parts of the work, marking the elements of this set as unusable only once rather than for each $f \in Cl$. This approach saves computation time as long as some of the clusters contain more than one event.

This way of computing possible extensions is fully compatible with preset trees construction described in Chapter 4 and, for some examples, it reduced the time needed for generating the prefix by more than 30%.

5.6 Experimental results

We used the unfolding algorithm described in Chapter 4 (with the cutting context Θ_{ERV}) as the basis for a parallel implementation and for the comparison. The experiments were conducted on a workstation with four *Pentium*^{TM} III/500MHz processors and 512M RAM. The parallel algorithm was implemented using POSIX threads. The methodology of testing correctness of the algorithm and the test cases are described in Chapter 3.

The results of experiments are summarized in Table 5.1. The meaning of the columns in the table is as follows (from left to right): the name of the problem; the number of places and transitions and the average/maximal size of transition presets in the original net system; the number of conditions, events and cut-off events in the complete prefix; the time spent by the sequential unfolder described in Chapter 4; the time spent by the parallel unfolder with different number N of working threads;[3] the average/maximal number of independent tasks which may be performed in parallel on each iteration of the main loop (this coincides with the number of clusters in CP). Although, due to the limited number of processors, we could not exploit all the arising parallelism in the performed experiments, this data shows the potential scalability of the problem.

It is interesting to note that the new algorithm with only one working thread ($N = 1$) works faster than the sequential unfolder described in Chapter 4. This is so due to the improvements discussed in Section 5.5.

One can see that the proposed algorithm does not achieve linear speedup. This was a surprising discovery, since the potential parallelism (the last column

[3] The concurrency relation was switched off for all these experiments.

in the table) is usually *very high*. Profiling shows that the program spends more than 95% of time in a function which neither acquires locks, nor performs system calls, so the contention on locks cannot be the reason for such a slowdown. The only rational explanation we could think of is bus contention: the mentioned function tries to find co-sets forming presets of possible extensions, exploring the built part of the prefix. It is a fairly large pointer-linked structure, and the processors have to intensively access the memory in a quite unsystematic way, so that the processors' caches often have to redirect the access to the RAM. Therefore, the processors are forced to contend for the bus, and the program slows down.

Since this explanation might seem superficial, we decided to establish that bus contention does reveal itself in practice, and the following experiment was performed. Several processors intensively read random locations in a large array and performed some fake computation with the fetched values. The total number of fetches was fixed and evenly distributed among them. In the absence of bus contention, the time spent by such a program would decrease linearly in the number of used processors, but we observed the degradation of speed similar to that shown by the unfolding algorithm. We expect that future generations of hardware will alleviate this problem, e.g., by increasing the bus frequency or by introducing a separate bus for each processor.

5.7 Conclusions

We proposed a new unfolding algorithm, which admits an efficient parallelization, both for shared and distributed memory architectures. Experimental results indicate that it can achieve significant (in theory, even linear) speedup. Moreover, due to the improved structure of the queue of possible extensions and the optimized routine for generating possible extensions, this algorithm is faster than the former implementations even in the sequential case.

Problem	Net		Unfolding			Time, [s]					
	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert B\rvert$	$\lvert E\rvert$	$\lvert E_{cut}\rvert$	*Seq*	*N=1*	*N=2*	*N=3*	*N=4*	$a/m\ \lvert CP\rvert$
Buf(100)	200	101	10101	5051	1	31	18	13	13	13	1.94/9
Byz	504	409	42276	14724	752	246	183	110	84	78	135.84/896
Dme(7)	470	343	9542	2737	49	7	5	2	2	1	42.02/56
Dme(8)	537	392	13465	3896	64	16	12	6	5	4	55.54/72
Dme(9)	604	441	18316	5337	81	33	26	14	11	10	71.03/90
Dme(10)	671	490	24191	7090	100	61	49	28	21	19	88.47/110
Dme(11)	738	539	31186	9185	121	105	86	50	39	35	107.89/132
Dph(6)	57	92	14590	7289	3407	10	7	3	3	2	62.05/127
Dph(7)	66	121	74558	37272	19207	286	211	126	97	90	219.96/509
Elev(4)	736	1939	32354	16935	7337	73	42	25	19	17	204.58/964
Ftp(1)	176	529	178085	89046	35197	2820	1609	975	761	714	915.57/3249
Furn(3)	53	99	30820	18563	12207	30	15	9	7	5	91.83/264
Gasnq(4)	258	465	15928	7965	2876	19	11	6	5	4	110.94/284
Gasnq(5)	428	841	100527	50265	18751	884	553	334	259	243	529.16/1400
Gasq(4)	1428	2705	19864	9933	4060	30	18	11	7	6	138.25/493
Key(3)	129	133	13941	6968	2911	10	7	4	3	2	57.91/145
Key(4)	164	174	135914	67954	32049	935	806	485	379	354	427.27/1224
Mmgt(3)	122	172	11575	5841	2529	6	4	2	1	1	96.17/328
Mmgt(4)	158	232	92940	46902	20957	556	339	205	159	150	567.77/1992
Q	163	194	16123	8417	1188	41	25	15	11	10	84.03/344
Rw(12)	63	313	98378	49177	45069	15	6	3	2	2	157.62/462
Sync(3)	106	270	28138	15401	5210	79	62	36	27	24	116.27/343
Rnd(5,8,500)	40	540	235600	56691	46559	68	51	29	22	19	386.81/1344
Rnd(5,9,500)	45	545	304656	72895	59840	113	90	53	41	37	447.62/1519
Rnd(5,10,500)	50	550	419946	98477	82279	175	144	85	66	61	474.97/1712
Rnd(5,11,500)	55	555	573697	132344	112310	267	227	134	104	99	526.03/1853
Rnd(5,12,500)	60	560	627303	145378	122465	351	297	178	140	131	557.76/1872
Rnd(5,13,500)	65	565	718762	166093	140147	453	382	232	183	172	539.40/1881
Rnd(5,14,500)	70	570	802907	185094	156417	546	471	284	225	215	584.35/1970
Rnd(5,15,500)	75	575	842181	195228	163722	665	567	345	274	259	605.35/1971
Rnd(5,16,500)	80	580	886158	206265	171957	787	674	413	329	312	623.24/2013
Rnd(5,17,500)	85	585	987605	229284	191576	942	822	503	404	382	607.82/2066
Rnd(5,18,500)	90	590	1025166	239069	198524	1091	956	584	469	448	614.02/2114
Rnd(10,3,500)	30	530	1415681	153628	144548	84	46	26	19	17	633.79/2095
Rnd(10,4,500)	40	540	2344821	252320	237000	216	137	80	61	55	720.00/2415
Rnd(10,5,500)	50	550	2485903	271083	250600	354	236	140	108	101	751.15/2406
Rnd(10,6,500)	60	560	2535070	280560	255010	526	360	216	168	159	746.97/2343
Rnd(10,7,500)	70	570	2537646	285323	254767	724	510	306	242	229	707.14/2323
Rnd(10,8,500)	80	580	2534970	289550	254000	953	681	411	327	312	786.64/2116
Rnd(15,2,500)	30	530	1836868	135307	128358	70	17	9	6	5	664.40/1979
Rnd(15,3,500)	45	545	3750719	271074	255560	270	128	74	56	49	895.59/2141
Rnd(15,4,500)	60	560	3787575	280560	257515	487	277	162	128	117	874.85/2301
Rnd(15,5,500)	75	575	3795090	288075	257515	776	480	286	228	214	819.19/2472
Rnd(20,2,500)	40	540	4744587	256197	245750	176	42	21	14	11	841.25/2797
Rnd(20,3,500)	60	560	5040080	280560	260020	447	203	118	90	82	842.21/2237
Rnd(20,4,500)	80	580	5050100	290580	260020	825	456	271	213	201	865.03/2510
Spa(7)	167	241	52516	18712	9937	81	48	28	21	19	169.27/629
Spa(8)	190	385	216772	76181	45774	1005	603	362	280	264	480.21/2002
Spa(9)	213	657	920270	320582	209449	13512	8066	4854	3750	3537	1669.04/6953
Spa(2,3)	144	161	15690	5682	2512	8	4	2	2	1	71.11/232
Spa(2,4)	190	385	253219	88944	52826	1412	872	524	406	382	614.64/2455
Spa(3,2)	144	161	15690	5682	2512	8	4	2	2	1	71.11/232
Spa(3,3)	213	657	1142214	398850	256600	22011	13565	8171	6317	5943	2166.84/8928

Table 5.1: Experimental results.

Chapter 6

Unfoldings of High-Level Petri Nets

The unfolding technique and algorithms described in Chapters 2, 4, and 5 help to alleviate the state space explosion problem when model checking low-level Petri nets. But their applicability may be considered restricted, since low-level Petri nets are a very low-level model, and thus inconvenient for practical modelling. Therefore, it is highly desirable to generalize this technique to more expressive formalisms, such as high-level (or 'coloured') Petri nets. This formalism allows one to model in quite a natural way many constructs of high-level specification languages used to describe concurrent systems (see, e.g., [5, 33, 34]). Though it is possible to translate a high-level net into a low-level one and then unfold the latter, it is often the case that the intermediate low-level net is exponentially larger than the resulting prefix. Moreover, such a translation often completely destroys the structure present in the original model.

In this chapter, we describe an approach which allows one to build a prefix directly from a high-level net, thus avoiding a potentially expensive translation into a low-level net. Experiments demonstrate that this method is often superior to the traditional one, involving the explicit construction of an intermediate low-level net. We show that it is possible to generate exactly the same prefix which would have been generated by the traditional approach, and so all the verification tools employing unfoldings can be re-used with prefixes generated by the method proposed in this chapter.

While writing up the original paper [58], it turned out that a related work had been reported in [68]. Therefore, we highlight here the main differences between the proposed approach and that of [68]. We establish an important relation between the branching processes of a high-level net and those of its low-level counterpart. This allows us to import the results of Chapters 2, 4, and 5 rather than re-prove them. Among such results are the canonicity of the prefix for different cutting contexts, the usability of the total adequate order $\lhd_{erv}$, and the parallel unfolding algorithm (neither of these were proved in [68]). Moreover, we adopt a different way of introducing branching processes of high-level nets, which

results in a neater and easier-to-comprehend presentation. In particular, we do not use algorithm-dependent proofs, and we tried to make all the definitions as similar to the corresponding ones for low-level nets as possible. Finally, we do not restrict ourselves to finite sets of colours, and fix a subtle mistake of [68] in the definition of cut-off events and the related prefix (see Remark 2.6).

This chapter is based on the results developed in [58, 59].

6.1 High-level Petri nets

In this chapter we use M-nets (see [4]) as the main high-level Petri net model, as we believe that it is general enough to cover many other existing relevant formalisms. The full description of M-nets can be found in [4]. Here, in order to match the presentation of low-level nets as closely as possible, we give suitably adapted short definitions, omitting those details which are not directly needed for the purposes of this chapter. In particular, [4] devotes a lot of attention to the composition rules for M-nets, which are relevant only at the construction stage of an M-net, but not for model checking of an already constructed one.

6.1.1 M-nets

It is assumed that there exists a (finite or infinite) set *Tok* of elements (or 'colours') and a set *VAR* of variable names, such that $\mathit{Tok} \cap \mathit{VAR} = \emptyset$. An *M-net* N is a quadruple $N \stackrel{\mathrm{df}}{=} (P, T, W, \iota)$ such that P and T are disjoint sets of respectively *places* and *transitions*, W is a multiset over $(P \times \mathit{VAR} \times T) \cup (T \times \mathit{VAR} \times P)$ of arcs, and ι is an inscription function with the domain $P \cup T$. It is assumed that, for every place $p \in P$, $\iota(p) \subseteq \mathit{Tok}$ is the *type* of p and, for every transition $t \in T$, $\iota(t)$ is a well-formed boolean expression over $\mathit{Tok} \cup \mathit{VAR}$, called the *guard* of t. We assume that the types of all places are finite.[1] In what follows, we assume that $N = (P, T, W, \iota)$ is a fixed M-net.

For a transition $t \in T$, let ${}^\bullet t \stackrel{\mathrm{df}}{=} \{\!|p^v \mid (p, v, t) \in W|\!\}$, $t^\bullet \stackrel{\mathrm{df}}{=} \{\!|p^v \mid (t, v, p) \in W|\!\}$, and $\mathit{VAR}(t) \stackrel{\mathrm{df}}{=} \{v \mid (p, v, t) \in W \vee (t, v, p) \in W\} \cup \mathit{VAR}(\iota(t))$, where $\mathit{VAR}(\iota(t))$ is the set of variables occurring in $\iota(t)$. (The notation p^v, similarly as p^x and t^σ used later on, is a shorthand for the pair (p, v).) A *firing mode* of t is a mapping $\sigma : \mathit{VAR}(t) \to \mathit{Tok}$ such that for all $p^v \in {}^\bullet t + t^\bullet$, $\sigma(v) \in \iota(p)$, and $\iota(t)$ evaluates to **true** under the substitution given by σ.

We define the set of *legal place instances* as $\mathcal{P} \stackrel{\mathrm{df}}{=} \{p^x \mid p \in P \wedge x \in \iota(p)\}$ and the set of *legal firings* as $\mathcal{T} \stackrel{\mathrm{df}}{=} \{t^\sigma \mid t \in T \text{ and } \sigma \text{ is a firing mode of } t\}$. For every $t^\sigma \in \mathcal{T}$, we will also denote ${}^\bullet t^\sigma \stackrel{\mathrm{df}}{=} \{\!|p^{\sigma(v)} \mid p^v \in {}^\bullet t|\!\}$ and $t^{\sigma\bullet} \stackrel{\mathrm{df}}{=} \{\!|p^{\sigma(v)} \mid p^v \in t^\bullet|\!\}$. According to the definitions given below, all valid markings of an M-net will be composed of legal place instances, and its firing sequences will be composed

[1] In general, allowing infinite types yields a Turing-powerful model. Nevertheless, this restriction can be omitted in certain important cases (see Section 6.4).

of legal firings. Furthermore, the sets $\mathcal{P}$ and $\mathcal{T}$ will provide the basis for the construction of the low-level net corresponding to a high-level one.

A *marking* M of N is a multiset over $\mathcal{P}$. We will denote the set of all such markings by $\mathcal{M}(N)$. (Traditionally, a marking is a mapping which, to every place $p \in P$, associates a multiset over $\iota(p)$. Clearly, such a representation is equivalent to the one we chose to use.)

The *transition relation* is a ternary relation on $\mathcal{M}(N) \times \mathcal{T} \times \mathcal{M}(N)$ such that a triple (M', t^σ, M'') belongs to it (denoted $M'[t^\sigma\rangle M''$) if ${}^\bullet t^\sigma \leq M'$ and $M'' = M' - {}^\bullet t^\sigma + t^{\sigma\bullet}$. Note that σ is a firing mode of t, which guarantees that M'' is a valid marking of N whenever M' was.

6.1.2 M-net systems

An *M-net system* is a pair $\Upsilon \overset{\text{df}}{=} (N, M_0)$ comprising a finite M-net N and an *initial marking* M_0. The set of *reachable markings* of an M-net system Υ is the smallest (w.r.t. $\subseteq$) set $\mathcal{RM}(\Upsilon)$ containing M_0 and such that if $M \in \mathcal{RM}(\Upsilon)$ and $M[t^\sigma\rangle M'$ in N, for some $t^\sigma \in \mathcal{T}$, then $M' \in \mathcal{RM}(\Upsilon)$.

An M-net system Υ is *k-bounded* if, for every marking $M \in \mathcal{RM}(\Upsilon)$ and every $p^x \in \mathcal{P}$, $M(p^x) \leq k$; *safe* if it is 1-bounded; and *bounded* if it is k-bounded for some $k \in \mathbb{N}$. Moreover, Υ is *strictly k-bounded* if, for every marking $M \in \mathcal{RM}(\Upsilon)$ and every place $p \in P$, $|\{\!|x \mid p^x \in M|\!\}| \leq k$, and *strictly safe* if it is strictly 1-bounded. One can show that strictly k-bounded M-net systems are k-bounded, strictly safe ones are safe, and the set $\mathcal{RM}(\Upsilon)$ is finite iff Υ is bounded. Note that according to the above definitions, a safe (but not strictly safe) M-net system can have a reachable marking which places several tokens on the same place, provided that their 'colours' are all distinct. The rational behind this choice of the definition is that the low-level *expansion* (defined below) of an M-net system is safe iff the original M-net system is safe, and so the total adequate order on configurations of unfoldings of safe net systems proposed in [30, 31, 85] can be re-used (see the end of Section 6.3).

We adopt the standard rules about drawing M-net systems, viz. places are represented as circles, transitions as boxes, and each element of W as a directed arc labelled by a variable. The values of the inscription function ι are indicated near the corresponding nodes, and markings are shown by placing tokens (e.g., natural numbers) within circles.

As an example, consider the M-net system shown in Figure 6.1(a). At the initial marking, t_1 can fire with the firing modes $\sigma \overset{\text{df}}{=} \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 1\}$ or $\sigma' \overset{\text{df}}{=} \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 2\}$, consuming the tokens from p_1 and p_2 and producing respectively the token 1 or 2 on p_3. Formally, we have $\{\!|p_1^1, p_2^2|\!\}[t_1^\sigma\rangle\{\!|p_3^1|\!\}$ and $\{\!|p_1^1, p_2^2|\!\}[t_1^{\sigma'}\rangle\{\!|p_3^2|\!\}$.

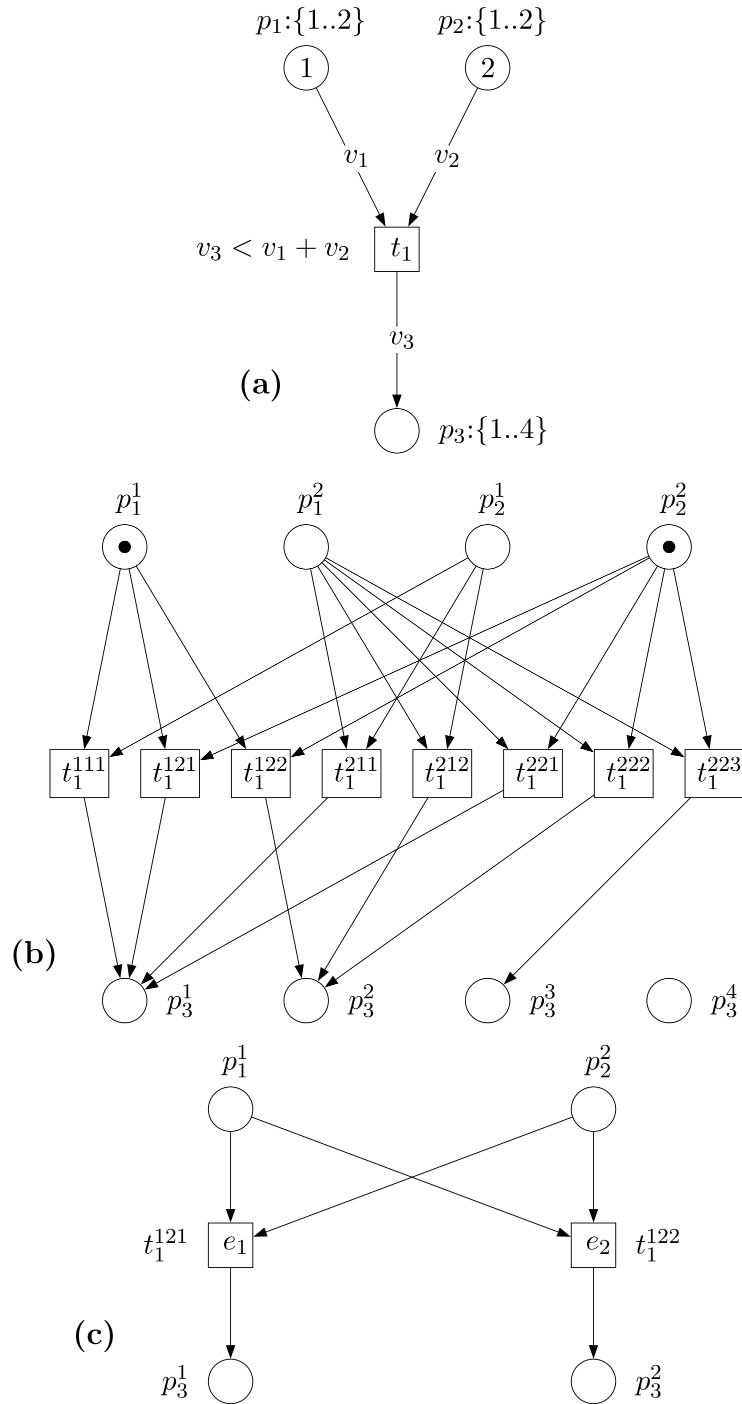

Figure 6.1: An M-net system **(a)**, its expansion **(b)**, and its unfolding **(c)**. Note that a firing mode σ of t is represented as a three-element string $\sigma(v_1)\sigma(v_2)\sigma(v_3)$.

6.2 Translation into low-level nets

For each M-net system it is possible to build an ‘equivalent’ low-level one. Such a transformation is called ‘unfolding’ in [4], but since we use this term in a different meaning (see Chapter 1), we adopt the term ‘expansion’ instead.

The *expansion* of an M-net $N = (P, T, W, \iota)$ is a low-level net $\mathcal{E}(N) \stackrel{\text{df}}{=} (\mathcal{P}, \mathcal{T}, W')$ where

$$W' \stackrel{\text{df}}{=} \sum_{t^\sigma \in \mathcal{T}} \left(\{\!\!| (p^{\sigma(v)}, t^\sigma) \mid (p, v, t) \in W |\!\!\} + \{\!\!| (t^\sigma, p^{\sigma(v)}) \mid (t, v, p) \in W |\!\!\} \right) .$$

Moreover, the expansion $\mathcal{E}(M)$ of a marking M of N is M itself, i.e., $\mathcal{E}(M) \stackrel{\text{df}}{=} M$ (this is possible since we deliberately chose the definitions so that the sets $\mathcal{M}(N)$ and $\mathcal{M}(\mathcal{E}(N))$ coincide). Finally, the expansion of an M-net system $\Upsilon = (N, M_0)$ is defined as $\mathcal{E}(\Upsilon) \stackrel{\text{df}}{=} (\mathcal{E}(N), \mathcal{E}(M_0))$. Figure 6.1(a,b) illustrates the last definition.

One can show that the following hold.

Proposition 6.1 ([4]). *Let N be an M-net, and $M', M'' \in \mathcal{M}(N)$. Then $M'[t^\sigma\rangle M''$ in Υ iff $M'[t^\sigma\rangle M''$ in $\mathcal{E}(\Upsilon)$.*

Proposition 6.2. *Let $\Upsilon = (N, M_0)$ be an M-net system.*

- *For every $k \in \mathbb{N}$, $\mathcal{E}(\Upsilon)$ is k-bounded iff Υ is k-bounded.*
- *$\mathcal{E}(\Upsilon)$ is safe iff Υ is safe.*
- *If Υ is strictly safe and p is a place of Υ, then the places p^x, $x \in \iota(p)$, are mutually exclusive in $\mathcal{E}(\Upsilon)$.*

Proof. Follows directly from the definitions. □

Though, according to Proposition 6.1, the expansion of an M-net system faithfully models the original system, the disadvantage of this transformation is that it typically yields a very large net. Moreover, the resulting net system is often *unnecessarily* large, in the sense that it contains many places which cannot be marked and many dead transitions. This is so because the place types are usually overapproximations, and the transitions of the original M-net system may have many firing modes, only few of which are realized when executing the net from the initial marking. For example, only two out of eight transitions of the expansion of the M-net system in Figure 6.1(a), shown in Figure 6.1(b), can actually fire. Therefore, though the M-net expansion is a neat theoretical construction, it is often impractical.

6.3 Branching processes of high-level Petri nets

In this section we develop the main results of this chapter, namely the notions of a branching process of an M-net system, the associated unfolding, and its canonical prefix. We also show that there is a strong correspondence between the branching processes of an M-net system and those of its expansion. This allows for importing many results from the theory of branching processes of low-level Petri nets.

Definition 6.3 (Branching Process of an M-net System). A *homomorphism* from an occurrence net $ON = (B, E, G)$ to an M-net system Υ is a mapping $h : B \cup E \to \mathcal{P} \cup \mathcal{T}$ such that

- $h(B) \subseteq \mathcal{P}$ and $h(E) \subseteq \mathcal{T}$ (conditions are mapped to legal place instances, and events to legal firings).
- For every $e \in E$, $h\{\!|{}^\bullet e|\!\} = {}^\bullet h(e)$ and $h\{\!|e^\bullet|\!\} = h(e)^\bullet$ (the environments of legal firings are preserved).
- $h\{\!|Min(ON)|\!\} = M_0$ (minimal conditions are mapped to the initial marking).
- For all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$, then $e = f$ (there is no redundancy).

A *branching process* of Υ is a pair $\pi \stackrel{\text{df}}{=} (ON, h)$ such that ON is an occurrence net and h is a homomorphism from ON to Υ. ◇

The above definition is illustrated in Figure 6.1.

Definition 6.3 coincides with the definition of a (low-level) branching process of $\mathcal{E}(\Upsilon)$. Thus most of the definitions for branching processes of low-level net systems can now be lifted to branching processes of M-net systems. In particular, this is the case for the notions of a *configuration*, *cut*, *final marking*, prefix relation $\sqsubseteq$, *cutting context*, and the notion of *completeness* of a prefix. Similarly, most of the results proven for branching processes of low-level Petri nets can also be lifted to branching processes of M-net systems. In particular, for each M-net system Υ there exist a unique (up to isomorphism) maximal w.r.t. $\sqsubseteq$ branching process Unf_Υ^{max} of Υ, called the *unfolding* of Υ. Moreover, for any cutting context Θ there exists unique *canonical* prefix Unf_Υ^Θ (coinciding with $Unf_{\mathcal{E}(\Upsilon)}^\Theta$) of Unf_Υ^{max}, and the theory of *canonical* prefixes (see Chapter 2) can be transferred without any changes.

Remark 6.4. One should be careful when dealing with adequate orders: though they are abstractly defined on the configurations of the unfolding, in practice the node labels are often employed in order to compute it. In particular, $\lhd$ is often parameterized by some order $\ll$ on the set of transitions of the low-level Petri net (see Chapter 2). Hence, in order to unfold an M-net system Υ one has to define such an order on $\mathcal{T}$ rather than on T. ◇

It is straightforward to give an upper bound on the size of Unf_{Υ}^{Θ}, since the results of Chapter 2 regarding the size of the canonical prefix are still applicable. In particular, for the cutting context Θ_{ERV} the number of non-cut-off events in Unf_{Υ}^{Θ} does not exceed $|\mathcal{RM}(\Upsilon)|$.

6.4 M-net unfolding algorithm

Due to the results developed in the previous section, it is now possible to modify the unfolding algorithms proposed in Chapters 2, 4, and 5 making them capable of building canonical prefixes of M-net unfoldings. It turns out that the only thing which has to be changed is the notion of a possible extension (so all the modifications are inside the PotExt (or UpdatePotExt) function and thus are not visible in the top-level description of the algorithm).

Definition 6.5. For a branching process π of an M-net system Υ, a *possible extension* is a pair (t^σ, D), where D is a co-set in π and $t^\sigma \in \mathcal{T}$ is a legal firing, such that $h\{\!|D|\!\} = {}^\bullet t^\sigma$ and π contains no t^σ-labelled event with preset D. $\diamond$

Similarly as in the low-level case, we will take the pair (t^σ, D) as a new event of the prefix, with the preset D. After it is inserted into the prefix, its postset D' consisting of new conditions such that $h\{\!|D'|\!\} = t^{\sigma\bullet}$ is also inserted.

It is worth noting that most of the existing heuristics aiming at speeding up the prefix generation can be applied. In particular, the total adequate order $\lhd_{erv}$ for safe net systems can be used to unfold safe M-net systems. (It remains adequate since Unf_{Υ}^{max} coincides with $Unf_{\mathcal{E}(\Upsilon)}^{max}$ and the expansion of a safe M-net system is safe.) Moreover, the *concurrency relation* (see [29, 85]) can also be employed, even for non-safe systems. As for the preset trees construction described in Chapter 4, it can be used without any modifications to unfold strictly safe M-net systems (and we work now on generalizing it to a wider class of M-net systems).

It turns out that directly unfolding a high-level net not only avoids the generation of its (potentially, very large) expansion, but often is also more efficient than unfolding its expansion. Indeed, as it was mentioned in Chapter 4, the most time-consuming part of the algorithm is computing the possible extensions of the built part of the prefix. Since one high-level transition usually corresponds to several low-level ones, less transitions have to be tried each time possible extensions are computed, which may lead to considerable savings in the running time. (Though the preset trees construction described in Chapter 4 alleviates the problem in the low-level case, it is not very efficient in the presence of many dead transitions.)

It is often the case that the information about the firing mode of an event needs not be explicitly stored. Indeed, this information almost always can be discarded, since one is usually not interested in what was the precise firing mode of a transition, as long as the consumed and produced tokens remain the same.

An important extension of the proposed approach allows for M-net systems with places having infinite types. For example, it is often convenient to assign to a place the type $\mathbb{N}$ rather than $\{0, \ldots, n\}$, since n might be not known in advance. Even when the set of reachable markings of such an M-net system is finite, its expansion is infinite and so of little use for model checking, whereas with the described direct approach we still can build the canonical prefix and complete the verification. The only thing which needs to be ensured is that at any stage of prefix construction only a finite number of legal firings needs to be considered. This will be the case if, for every transition t and every finite multiset Z over $\mathcal{P}$, the set of all firing modes σ of t such that ${}^\bullet t^\sigma \leq Z$ is both finite and computable.

Having built a canonical prefix, one can easily construct the refined version of the low-level expansion of the original M-net system, with unreachable places and dead transitions removed. This may be important, e.g., for directly mapping a Petri net to a circuit simulating its behaviour.

Finally, it is worth mentioning that since the proposed method constructs exactly the same prefix which would have been generated from the corresponding expansion of the M-net system, all the existing model checkers employing unfolding prefixes derived from low-level nets can be used without any changes when dealing with prefixes generated directly from M-net systems.

6.5 Case studies

In this section, we compare the described approach with the traditional one, viz. the unfolding of M-net expansions. We used the unfolding engine described in Chapters 4 and 5 which after suitable modifications was able to unfold both low-level and high-level Petri nets. For building M-net expansions, we used the HL2LL utility from the PEP tool (see [6,7]).

The meaning of the columns in Tables 6.1 and 6.2 is as follows (from left to right): the name of the problem; the number of places and transitions in the original M-net system; the number of places and transitions in the corresponding expansion, together with the time required by the HL2LL utility to build the expansion; the number of conditions, feasible events, and cut-off events in the canonical prefix; the time (in seconds) required to unfold the expansion of the M-net system and the M-net system itself, respectively.

The first example is data-intensive, and so the traditional (via low-level nets) approach is extremely inefficient, whereas we expected the new algorithm to perform well. The second example is control-intensive, so the M-net expansions are just slightly larger that the original M-nets. It was chosen to test the worst-case performance of the proposed method relatively to the unfolding of the low-level expansion.

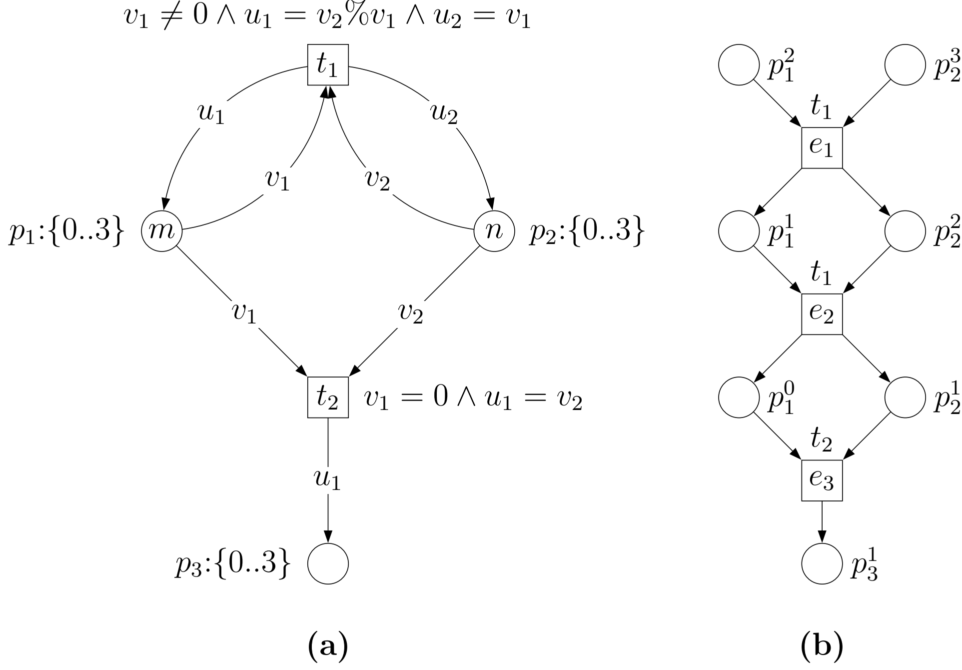

Figure 6.2: An M-net system modelling Euclid's algorithm for computing the greatest common divisor of two non-negative integers **(a)** and its unfolding for $m = F_3 = 2$ and $n = F_4 = 3$ **(b)** (firing modes are not shown).

6.5.1 Greatest common divisor

An M-net simulating Euclid's algorithm for computing the greatest common divisor of two non-negative integers m and n and its unfolding are shown in Figure 6.2. This M-net is very data-intensive, and thus its expansion is much larger than the original high-level net, especially when the values of m and n are large. In these experiments, we computed the greatest common divisor of two consecutive Fibonacci's numbers, F_i and F_{i-1}, for different values i (such numbers are known to produce the longest sequences of computational steps for Euclid's algorithm). The results of these experiments are summarized in Table 6.1. From the structure of the M-net it is easy to calculate that its expansion contains $3(F_i+1)$ places and $(F_i+1)^2$ transitions (note that F_i is exponential in i). These values are reported in the corresponding columns of the table, even though HL2LL failed to produce the expansions when they became large.

The experimental results show that for this example the high-level unfolding is clearly superior. Though the M-net expansion grows very quickly, the resulting prefix has only $2i - 1$ conditions and $i - 1$ events. Therefore, the new algorithm was able to build it for relatively large i (we had to stop the experiments after $i = 45$ since F_{50} overflows 4-bytes integer, but it is a limitation of the current implementation rather than of the method itself).

Problem	M-net		Expansion			Unfolding			t[s]	
	$\|P\|$	$\|T\|$	$\|P\|$	$\|T\|$	t[s]	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	LL	HL
GCD(F_5, F_4)	3	2	18	36	<1	9	4	0	<1	<1
GCD(F_{10}, F_9)	3	2	168	3136	1	19	9	0	6	<1
GCD(F_{15}, F_{14})	3	2	1833	$>10^5$	—	29	14	0	—	<1
GCD(F_{20}, F_{19})	3	2	$>10^4$	$>10^7$	—	39	19	0	—	<1
GCD(F_{25}, F_{24})	3	2	$>10^5$	$>10^9$	—	49	24	0	—	<1
GCD(F_{30}, F_{29})	3	2	$>10^6$	$>10^{11}$	—	59	29	0	—	<1
GCD(F_{35}, F_{34})	3	2	$>10^7$	$>10^{13}$	—	69	34	0	—	<1
GCD(F_{40}, F_{39})	3	2	$>10^8$	$>10^{16}$	—	79	39	0	—	<1
GCD(F_{45}, F_{44})	3	2	$>10^9$	$>10^{18}$	—	89	44	0	—	<1

Table 6.1: Experimental results for M-net systems modelling Euclid's algorithm.

6.5.2 Mutual exclusion algorithm

The previous example was rather favourable for the new algorithm, since the expansions of the M-net systems were very large. We therefore checked the performance of the proposed approach in a totally opposite case, when the expansion of an M-net is relatively small. This happens when the transitions of the M-net are connected to few places and the cardinality of most place types is 1. Such M-nets arise when modelling Lamport's mutual exclusion algorithm for n processes trying to access a critical section (see [49, 70]). Its distinctive characteristic is 'very small' atomic actions. The pseudo-code of this algorithm is shown in Figure 6.3. We encoded it in the $B(PN)^2$ language supported by the PEP tool (see [5, 7, 33, 34]), as shown in Figure 6.4. Note that we had to replicate parts of the code since currently $B(PN)^2$ does not support the **goto** operator. The PEP tool can automatically compile a $B(PN)^2$ program into both a high-level and a low-level Petri net.

The type of the places corresponding to the variables x and y is $\{0, \ldots, n\}$, the type of places corresponding to b_i's is $\{\mathbf{false}, \mathbf{true}\}$, and all the other places have the type $\{\bullet\}$ and thus are not replicated in the expansion. Every transition has not more than 2 incoming and 2 outgoing arcs, and is connected to at least two places of type $\{\bullet\}$; moreover, in all assignments and conditions, one of the operands is always a constant. Therefore, the number of transition replicas in the expansion is relatively small.

The experimental results for Lamport's mutual exclusion algorithm are shown in Table 6.2. As one can see, the new algorithm performs almost as well as the algorithm for low-level nets. Though there is some overhead when computing transition guards and more complicated final states, it is relatively small, because the most time-consuming operation is computing the possible extensions of a current prefix. Moreover, this overhead becomes relatively smaller as the size of the prefix grows (it is just 0.5% for the last example in the table).

```
start :
   < b_i ← true >;
   < x ← i >;
   if < y ≠ 0 > then
      < b_i ← false >;
      await < y = 0 >;
      goto start;
   < y ← i >;
   if < x ≠ i > then
      < b_i ← false >;
      for j ← 1 to n do
            await < b_j = false >;
      if < y ≠ i > then
            await < y = 0 >;
            goto start;

   critical section;

   < y ← 0 >;
   < b_i ← false >;
```

Figure 6.3: The pseudocode of the i-th process in Lamport's mutual exclusion algorithm.

After the prefixes had been built, we verified using the efficient model checker described in Chapter 7 that the M-net system is deadlock free, and that the places corresponding to the critical sections of the processes are mutually exclusive. This was done without recompiling the model checker, since the new unfolding algorithm generates prefixes which are indistinguishable from those generated by a low-level net unfolder from the corresponding expansions of the M-nets.

It is worth noting that in this example partial-order methods have advantage over the state-space based ones. In [49], this mutual exclusion algorithm was verified for $n = 3$ by building a reachability graph of the Petri net model and for $n = 4$ by applying symmetry reductions. We managed to verify the case $n = 4$ without applying symmetry reductions, using a PC with smaller memory (128M rather than 256M), for a net system which was generated from a relatively high-level description ($B(PN)^2$ language) rather than built by hand. Moreover, as it was already noted, the specification we used was not optimal since we had to replicate parts of the code. In principle, due to the results developed in Chapter 2, it is also possible to apply partial-order methods together with symmetry reductions (see also [24]) to achieve even better results, but we have not implemented the combined method yet.

```
begin

VAR x : {0..n} init 0;
VAR y : {0..n} init 0;
VAR b1, ..., bn : {false, true} init false;

proc PROCESS(const i : {1..n}, ref b : {false, true}) _max_ n
begin
   do
      < b' = true >;  < x' = i >;
      do
         < y # 0 >;  < b' = false >;  < y = 0 >; exit
      □  < y = 0 >;  < y' = i >;
         do
            < x#i >;  < b' = false >;
            < b1 = false >;  ...;  < bn = false >;
            do
               < y#i >;  < y = 0 >; exit
            □  < y = i >;

               critical section

               < y' = 0 >;  < b' = false >; exit
            od; exit
         □  < x = i >;

            critical section

            < y' = 0 >;  < b' = false >; exit
         od; exit
      od; repeat
   od
end ;

PROCESS(1, b1) || ... || PROCESS(n, bn)

end
```

Figure 6.4: The $B(PN)^2$ code for Lamport's mutual exclusion algorithm.

Problem	M-net		Expansion			Unfolding			t[s]	
	$\|P\|$	$\|T\|$	$\|P\|$	$\|T\|$	t[s]	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	LL	HL
LAMP(2)	52	50	58	88	<1	711	368	102	<1	<1
LAMP(3)	77	76	86	154	<1	23424	12026	4562	29	30
LAMP(4)	104	104	116	236	<1	736507	375983	167780	28772	28917

Table 6.2: Experimental results for M-net systems modelling Lamport's mutual exclusion algorithm.

6.6 Conclusions

In this chapter, we defined branching processes and unfoldings of high-level Petri nets and proposed an algorithm which builds finite and complete prefixes of such unfoldings. We established an important relation between the branching processes of a high-level net and those of its low-level expansion. This allows for importing results proven for branching processes of low-level nets rather than re-prove them. Among such results are the canonicity of the prefix for different cutting contexts, the usability of the total adequate order $\lhd_{erv}$, and the parallel unfolding algorithm. The proposed approach is conservative in the sense that all the verification tools employing the traditional unfoldings can be reused with such prefixes. The conducted experiments demonstrated that it is, on one hand, superior to the traditional approach on data-intensive application, and, on the other hand, has the same performance on control-intensive ones.

Chapter 7

Prefix-Based Model Checking

Chapters 4–6 discussed how to efficiently generate finite and complete prefixes of Petri net unfoldings. In this chapter, we show how such prefixes can be used for efficient model checking reachability-like properties, i.e., finding reachable states satisfying certain properties, e.g., deadlocked markings.

In [77], the problem of deadlock checking a Petri net was reduced to a mixed integer linear programming (*MIP*) problem. In this chapter, we present a further development of this approach. We adopt Contejean and Devie's algorithm (*CDA*), developed in [1, 2, 16–18], for efficiently solving systems of linear constraints over the domain of natural numbers, and refine it by employing specific properties of the systems of linear constraints to be solved. The essence of the proposed modifications is to transfer the information about causality and conflicts between events involved in an unfolding into a relationship between the corresponding integer variables in the system of linear constraints. Experimental results demonstrate that the resulting algorithms can achieve significant speedups.

This chapter is based on the results developed in [52–55].

7.1 Deadlock detection using linear programming

In the rest of this chapter, we assume that Θ is a dense cutting context with the equivalence relation refining $\approx_{mar}$, and $\Sigma^\Theta \stackrel{\mathrm{df}}{=} (B, E, G, M_{in})$ is the safe net system built from the canonical prefix $Unf_\Sigma^\Theta = (B, E, G, h)$ of the unfolding of a bounded net system $\Sigma = (P, T, F, M_0)$, where M_{in} is the canonical initial marking of Σ^Θ which places a single token in each of the minimal conditions and no token elsewhere.[1] Furthermore, we will assume that $b_1, b_2, \ldots, b_p$ and $e_1, e_2, \ldots, e_q$ are respectively the conditions and events of Unf_Σ^Θ, and that $\mathfrak{I}_{\Sigma^\Theta}$ is the $p \times q$ incidence matrix of Σ^Θ. The set of cut-off events of Unf_Σ^Θ will be denoted by E_{cut}.

[1] We will often identify Σ^Θ and Unf_Σ^Θ, provided that this does not create an ambiguity.

We now recall the main results from [77]. Since Unf_{Σ}^{Θ} is complete, each reachable deadlocked marking in Σ is represented by a deadlocked marking in Σ^{Θ}. However, Σ^{Θ} can have additional deadlocks introduced by truncating the unfolding of Σ. Such false deadlocks can be eliminated by prohibiting the cut-off events from occurring.

Since for an acyclic Petri net the feasibility of the marking equation (see Section 1.4) is a sufficient condition for a marking to be reachable, the problem of deadlock checking can be reduced to the feasibility test of a system of linear constraints.

Proposition 7.1 ([77]). *Σ is deadlock-free iff the following system has no solution (in M and x):*

$$\begin{cases} M = M_{in} + \mathfrak{I}_{\Sigma^{\Theta}} \cdot x \\ \sum_{b \in {}^{\bullet}e} M(b) \leq |{}^{\bullet}e| - 1 \quad \textit{for all } e \in E \\ x(e) = 0 \quad \textit{for all } e \in E_{cut} \\ M \in \mathbb{N}^p \textit{ and } x \in \mathbb{N}^q \ , \end{cases} \tag{7.1}$$

where $x(e_i) = x_i$, for every $i \in \{1, \ldots, q\}$.

The first equation in (7.1) is the marking equation for Σ^{Θ} in the matrix form (see Section 1.4); it states that M is a marking of Σ^{Θ} reachable via some sequence of transitions whose Parikh vector is x. The second set of equations states that M enables no transitions of Σ^{Θ} and thus is a deadlocked marking (note that Σ^{Θ} is a safe net system). The last set of equations requires all the cut-off events not to occur in the execution sequence leading to M, and thus excludes the 'false' deadlocks from the set of solutions of the system.

In order to decrease the number of integer variables, $M \geq \mathbf{0}$ can be treated as a rational vector, since $x \in \mathbb{N}^q$ and $M = M_{in} + \mathfrak{I}_{\Sigma^{\Theta}} \cdot x \geq \mathbf{0}$ always imply that $M \in \mathbb{N}^p$. Moreover, as an event can occur at most once in a given execution sequence of Σ^{Θ} from the initial marking M_{in}, it is possible to require x to be a binary vector, $x \in \{0,1\}^q$.

To solve the resulting mixed integer LP-problem (*MIP* problem), [77] used the general-purpose LP-solver CPLEX^{TM} [23], and demonstrated that there are significant performance gains if the number of cut-off events is relatively high, since all variables in x corresponding to cut-off events are set to 0.

We will show in Section 7.3 that it is possible to reduce (7.1) to a purely integer LP-problem without increasing the total number of integer variables. Moreover, (7.1) has several problem-specific interdependencies between the variables, and taking them into account may allow one to significantly reduce the time needed to solve the system. Therefore, it turns out to be non-optimal to use general-purpose LP-solvers for this particular problem.

7.2 Solving systems of linear constraints

In this chapter, we will adapt the approach proposed in [1, 2, 16–18], in order to solve Petri net verification problems which can be reformulated as LP-problems. We start by recalling some basic results.

The original *Contejean and Devie's algorithm*, or *CDA* ([16–18]), solves a system of linear homogeneous equations with arbitrary integer coefficients

$$\begin{cases} \mathcal{A}_{11}x_1 & + & \cdots & + & \mathcal{A}_{1q}x_q & = & 0 \\ \mathcal{A}_{21}x_1 & + & \cdots & + & \mathcal{A}_{2q}x_q & = & 0 \\ \vdots & & & & \vdots & & \vdots \\ \mathcal{A}_{p1}x_1 & + & \cdots & + & \mathcal{A}_{pq}x_q & = & 0\,, \end{cases} \tag{7.2}$$

or $\mathcal{A} \cdot x = \mathbf{0}$, where $x \in \mathbb{N}^q$ and $\mathcal{A} \stackrel{\text{df}}{=} (\mathcal{A}_{ij})$. For every $1 \leq j \leq q$, let

$$\varepsilon_j \stackrel{\text{df}}{=} (\underbrace{0, \ldots, 0,}_{j-1 \text{ times}} 1, 0, \ldots, 0)$$

be the j-th vector in the canonical basis $\mathcal{CB}$ of $\mathbb{N}^q$. Vector $\mathcal{A} \cdot \varepsilon_j$ — the j-th column vector of the matrix $\mathcal{A}$ — is called the *j-th basic default vector*.

The set $\mathcal{S}$ of all solutions of (7.2) can be represented by a finite basis $\mathcal{BS}$ which is the minimal (w.r.t. $\subseteq$) subset of $\mathcal{S}$ such that every solution is an $\mathbb{N}$-linear combination of the solutions in $\mathcal{BS}$. It can be shown that $\mathcal{BS}$ comprises all solutions in $\mathcal{S}$ different from the trivial one, $x = \mathbf{0}$, which are minimal with respect to the $\leq$ ordering on $\mathbb{N}^q$ ($x \leq x'$ if $x_i \leq x'_i$, for all $1 \leq i \leq q$; moreover, $x < x'$ if $x \leq x'$ and $x \neq x'$).

Any solution of (7.2) can be seen as a multiset of default vectors whose sum is $\mathbf{0}$. Choosing an arbitrary order among these vectors amounts to constructing a sequence of default vectors starting from, and returning to, the origin of $\mathbb{Z}^p$. *CDA* constructs such a sequence step by step: starting from the empty sequence, new default vectors are added until a solution is found, or no minimal solution can be obtained. However, different sequences of default vectors may correspond to the same solution (up to permutation of vectors). To eliminate some of the redundant sequences, a restriction for choosing the next default vector is used.

Branching Condition 1. *A vector $x \in \mathbb{N}^q$ (corresponding to a sequence of default vectors) such that $\mathcal{A} \cdot x \neq \mathbf{0}$ can be incremented by 1 on its j-th component provided that $\mathcal{A} \cdot (x + \varepsilon_j) = \mathcal{A} \cdot x + \mathcal{A} \cdot \varepsilon_j$ lies in the half-space containing $\mathbf{0}$ and delimited by the affine hyperplane perpendicular to the vector $\mathcal{A} \cdot x$ at its extremity when originating from $\mathbf{0}$ (see Figure 7.1).* ◇

This reflects a view that $\mathcal{A} \cdot x$ should not become too large, hence adding $\mathcal{A} \cdot \varepsilon_j$ to $\mathcal{A} \cdot x$ should yield a vector $\mathcal{A} \cdot (x + \varepsilon_j) = \mathcal{A} \cdot x + \mathcal{A} \cdot \varepsilon_j$ 'returning to the origin'. Formally, this restriction can be expressed as

$$(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_j) < 0\,, \tag{7.3}$$

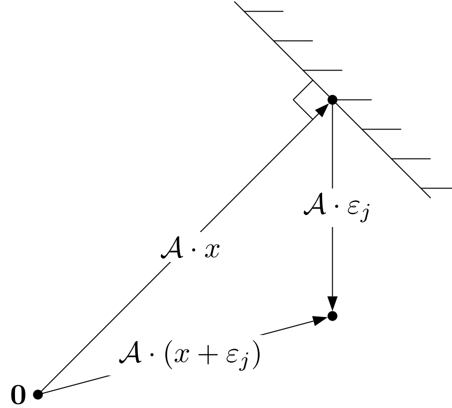

Figure 7.1: Geometric interpretation of the branching condition in *CDA*.

where $\odot$ denotes the scalar product of two vectors.

This reduces the search space without losing any minimal solution, since every sequence of default vectors which corresponds to a solution can be rearranged into a sequence satisfying (7.3).

Proposition 7.2 (Correctness, [18]). *The following hold for* CDA *shown in Figure 7.2:*

1. *Every minimal solution of (7.2) is found.* *(completeness)*
2. *Every solution found by* CDA *is minimal.* *(soundness)*
3. *The algorithm always terminates.* *(termination)*

> search breadth-first a directed acyclic graph rooted at $\varepsilon_1, \ldots, \varepsilon_q$
> **if** a node y is equal to, or greater than, an already found
>   solution of $\mathcal{A} \cdot x = \mathbf{0}$
> **then** y is a terminal node
> **else** construct the sons of y by computing $y + \varepsilon_j$
>   for each $j \leq q$ satisfying $\mathcal{A} \cdot y \odot \mathcal{A} \cdot \varepsilon_j < 0$

Figure 7.2: An outline of *CDA* (breadth-first version).

Figure 7.3(a) illustrates the process of solving the homogeneous system of linear equations

$$\begin{cases} - x_1 + x_2 + 2x_3 - 3x_4 = 0 \\ - x_1 + 3x_2 - 2x_3 - x_4 = 0 \,, \end{cases}$$

considered in [69]. The example shows redundancies, as some vectors were computed more than once. This can be remedied by using *frozen components*, defined

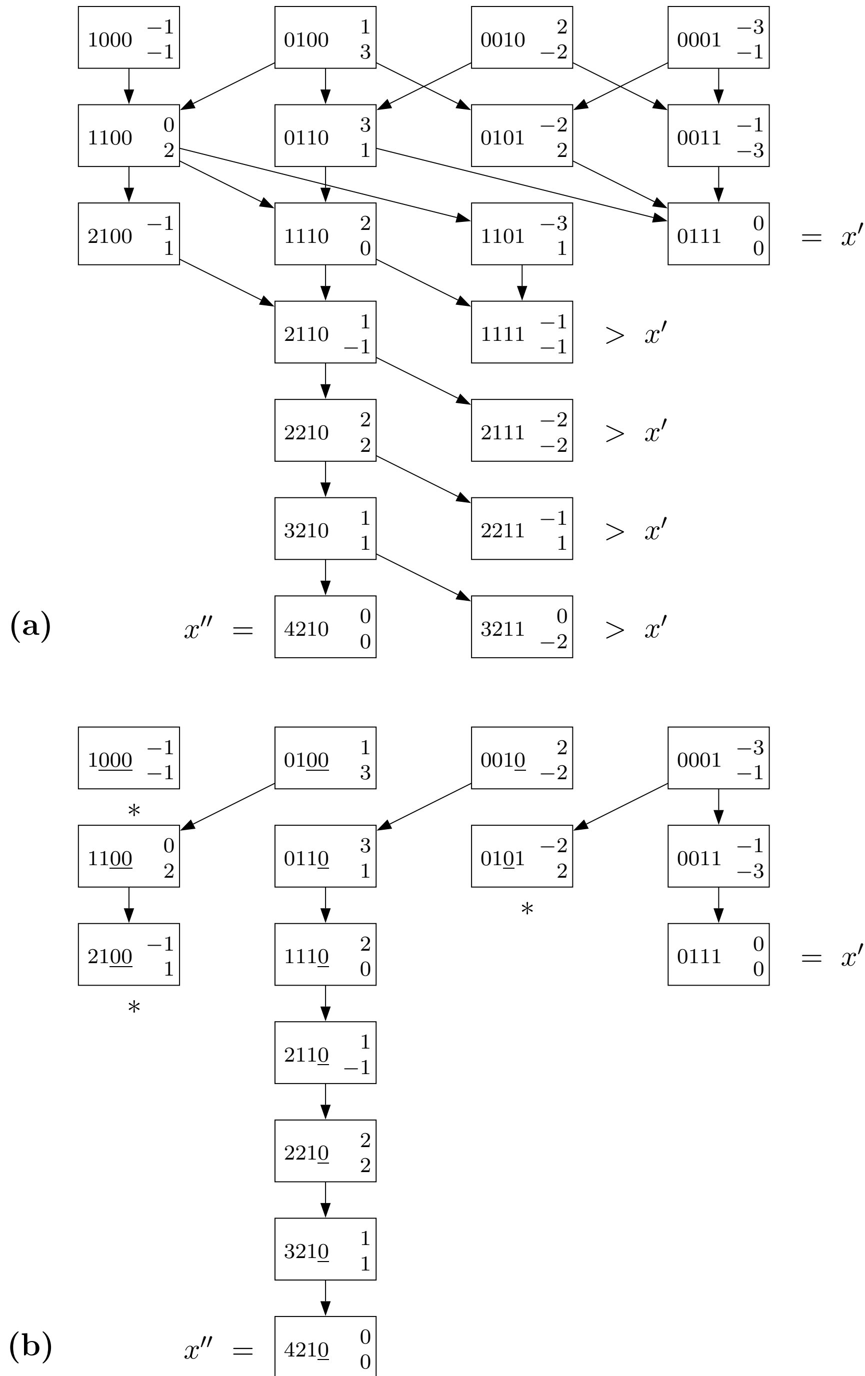

Figure 7.3: Search graphs constructed by the breadth-first **(a)** and ordered **(b)** versions of *CDA*. Inside each box, the current value of $\mathcal{A} \cdot x$ is represented by a column on the right, and is preceded by the current value of x. In the latter graph, frozen components are underlined, and the *s indicate the nodes which cannot be developed due to branching condition (7.3) and/or the frozen components rule. Note that $x' = (0, 1, 1, 1)$ and $x'' = (4, 2, 1, 0)$ are two minimal solutions.

thus. Assume that there is a total ordering $\prec_x$ on the sons of each node[2] x of the search graph constructed by *CDA*.

Frozen Components 1. *If $x + \varepsilon_i$ and $x + \varepsilon_j$ are two distinct sons of a node x such that $x + \varepsilon_i \prec_x x + \varepsilon_j$, then the j-th component is* frozen *in the sub-graph rooted at $x + \varepsilon_i$ and cannot be incremented even if (7.3) is satisfied.* $\diamond$

The modified algorithm is still complete (see [18]), and builds a forest which is a sub-graph of the original search graph. By defining[3] the ordering $\prec_x$ as $x + \varepsilon_i \prec_x x + \varepsilon_j \;\Leftrightarrow\; i < j$ we obtain, for the system in the above example, the graph shown in Figure 7.3(b) (see [69]).

The ordered version of *CDA* can easily handle bounds imposed on variables:

- $x' \leq x$. Then, instead of starting with the vectors $\varepsilon_1, \ldots, \varepsilon_q$, the algorithm starts with x'. The rest of the operation remains the same, but the minimal elements of the set $\mathcal{S}' = \{x \mid \mathcal{A}\cdot x = \mathbf{0} \wedge x' \leq x\}$ do not give all the solutions of
$$\begin{cases} \mathcal{A}\cdot x = \mathbf{0} \\ x' \leq x\ . \end{cases}$$
However, any solution of the above system can be represented as a sum of a minimal element of $\mathcal{S}'$ and an $\mathbb{N}$-linear combination of minimal solutions of the original system.

- $x \leq x''$ where $x'' \in (\mathbb{N} \cup \{\infty\})^q$. Then the algorithm works in the standard way except that the j-th component of a vector becomes frozen as soon as it reaches the j-th component of x''.

- $x' \leq x \leq x''$. Then a combination of the two previous techniques is used.

With the above extensions, *CDA* allows one to solve non-homogeneous diophantine systems

$$\begin{cases} \mathcal{A}_{11}x_1 & + & \cdots & + & \mathcal{A}_{1q}x_q & = & \alpha_1 \\ \mathcal{A}_{21}x_1 & + & \cdots & + & \mathcal{A}_{2q}x_q & = & \alpha_2 \\ \vdots & & & & \vdots & & \vdots \\ \mathcal{A}_{p1}x_1 & + & \cdots & + & \mathcal{A}_{pq}x_q & = & \alpha_p\ . \end{cases} \tag{7.4}$$

By introducing a new variable, x_0, one can transform (7.4) into a homogeneous system

$$\begin{cases} -\alpha_1 x_0 & + & \mathcal{A}_{11}x_1 & + & \cdots & + & \mathcal{A}_{1q}x_q & = & 0 \\ -\alpha_2 x_0 & + & \mathcal{A}_{21}x_1 & + & \cdots & + & \mathcal{A}_{2q}x_q & = & 0 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ -\alpha_p x_0 & + & \mathcal{A}_{p1}x_1 & + & \cdots & + & \mathcal{A}_{pq}x_q & = & 0\ . \end{cases}$$

[2] Including the virtual node $\mathbf{0}$.

[3] The ordering $\prec_x$ may be defined in other ways as well (see [18]).

Let $\mathcal{BS}_k$ ($k = 0, 1$) be the set of all minimal solutions $x = (x_0, x_1, \ldots, x_q)$ of this system with $x_0 = k$. Then any solution of (7.4) can be represented as

$$x = y + \sum_{z \in \mathcal{BS}_0} c_z \, z \; ,$$

where $y \in \mathcal{BS}_1$ and each c_z belongs to $\mathbb{N}$. Thus, to solve (7.4), it suffices to add just one variable which becomes frozen as soon as it reaches the value 1.

The task of solving a system of linear inequalities

$$\begin{cases} \mathcal{B}_{11}x_1 & + & \cdots & + & \mathcal{B}_{1q}x_q & \leq & \beta_1 \\ \mathcal{B}_{21}x_1 & + & \cdots & + & \mathcal{B}_{2q}x_q & \leq & \beta_2 \\ \quad\vdots & & & & \vdots & & \vdots \\ \mathcal{B}_{p1}x_1 & + & \cdots & + & \mathcal{B}_{pq}x_q & \leq & \beta_p \end{cases} \tag{7.5}$$

is more complicated. In general, not all the solutions of (7.5) can be represented as $\mathbb{N}$-linear combinations of minimal solutions, even if the system of inequalities is homogeneous. As an example, [2] considers the inequality $x_1 - x_2 \leq 0$. Its only non-trivial minimal solution is $(0, 1)$, which is not enough to generate the set of all solutions, $\{(n, n+m) \mid n, m \in \mathbb{N}\}$. To generate the whole set one needs also to take a non-minimal solution $(1, 1) > (0, 1)$.

The standard linear programming approach is to reduce (7.5) to a system of equations

$$\begin{cases} \mathcal{B}_{11}x_1 & + & \cdots & + & \mathcal{B}_{1q}x_q & + & y_1 & & & & = & \beta_1 \\ \mathcal{B}_{21}x_1 & + & \cdots & + & \mathcal{B}_{2q}x_q & + & & y_2 & & & = & \beta_2 \\ \quad\vdots & & & & \vdots & & & & \ddots & & & \vdots \\ \mathcal{B}_{p1}x_1 & + & \cdots & + & \mathcal{B}_{pq}x_q & + & & & & y_p & = & \beta_p \end{cases}$$

by introducing *slack variables* $y_i \in \mathbb{N}$, but this transformation increases the number of variables from q to $q+p$. Consequently, as the computation time can grow exponentially in the number of variables, such an approach is not efficient. Moreover, the slack variables may assume arbitrary values in $\mathbb{N}$, even if all the variables in the original problem are binary as in (7.1); as a result, the search space can grow very rapidly.

Another approach is to deal with the inequalities (7.5) directly. It was developed in [1,2], where *CDA* has been generalized to solve homogeneous systems of inequalities. The approach uses the notion of a *non-decomposable* solution, i.e., one which cannot be represented as an $\mathbb{N}$-linear combination of other solutions; one can see that the non-decomposable solutions form a basis of the set of all the solutions. For a system of linear constraints $\mathcal{A} \cdot x = \mathbf{0} \;\wedge\; \mathcal{B} \cdot x \leq \mathbf{0}$, the branching condition (7.3) is modified in the following way.

Branching Condition 2. *Given a vector $x = (x_1, \ldots, x_q)$, increment by* 1 *an element x_j for which there exist $y_1, \ldots, y_p$ such that the vector $(x_1, \ldots, x_q,$*

$y_1, \ldots, y_p$) can be incremented on its j-th component according to (7.3) applied to the system $\mathcal{A} \cdot x = \mathbf{0} \;\wedge\; \mathcal{B} \cdot x + y = \mathbf{0}$, where p is the number of rows in $\mathcal{B}$ and $y = (y_1, \ldots, y_p)$. ◇

As shown in [1,2], this condition can be expressed as

$$(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{p} \min \left\{ \begin{array}{l} (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j), \\ \max\{0, \mathcal{B}_i \odot x\}(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} < 0 \,, \tag{7.6}$$

where $\mathcal{B}_i$ is the i-th row of $\mathcal{B}$. To ensure termination in the general case, [1,2] add one more condition, but if all the variables are bounded (which is the case for all applications considered in this chapter) then this is not necessary.

7.3 Integer programming verification algorithm

In this section we start by reformulating the deadlock detection problem — one of the fundamental verification problems for Petri nets — in terms of integer programming. We then describe how solving the obtained system of constraints can be improved by taking into account partial-order dependencies between the variables derived from the unfolding. After that we develop an extension of *CDA* aimed at combining these dependencies with the original algorithm.

7.3.1 Reduction to a pure integer problem

The problem recalled in Section 7.1 can be reduced to a pure integer one, by substituting the expression for M given by the marking equation into the other constraints. Each equation in $M = M_{in} + \mathfrak{I}_{\Sigma^\Theta} \cdot x$ has the form

$$M(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \,, \tag{7.7}$$

where $b \in B$. After substituting these into (7.1) we obtain the system

$$\begin{cases} \displaystyle\sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) \text{ for all } e \in E \\ M_{in} + \mathfrak{I}_{\Sigma^\Theta} \cdot x \geq \mathbf{0} \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \,. \end{cases} \tag{7.8}$$

Usually, each inequality in (7.8) contains relatively few variables, so it does make sense to use a sparse-matrix representation of this system.

Remark 7.3. For efficiency reasons, inequalities can be first generated without paying attention to possible repetitions of the same variable in its left-hand side, and then sorted and transformed into the normal form. But one should be careful when choosing the sorting algorithm: the sequence of monomials obtained after generating the inequalities is often nearly sorted, and QUICKSORT performs rather poorly, i.e., in quadratic time. Experiments conducted at the initial stage of this work showed that in this case the process of sorting monomials can be much more time consuming than the process of solving the system; it is therefore better to use a sorting algorithm with $O(n \log n)$ worst case execution time. In the final implementation, we obtained satisfactory results with HEAPSORT, which has an additional advantage that it does not require auxiliary arrays. Alternatively, one can generate the equations on-the-fly, as described in Section 7.9.

As (7.8) is a pure integer problem, the usual integer programming algorithms are in principle directly applicable. However, since the number of variables is usually large even for moderate sized net systems, a further refinement is needed.

7.3.2 Partial-order dependencies between variables

In [77], Σ^Θ is used only for building a system of constraints, and the latter is then passed to an LP-solver without any additional information. Yet, while solving the system, one can use dependencies between variables implied by the causal order on events, which can easily be derived from Σ^Θ. For example, if we set $x(e) = 1$ then each $x(f)$ such that f is a predecessor (in the causal order) of e must be equal to 1, and each $x(g)$ such that g is in conflict with e, must be equal to 0. Similarly, if we set $x(e) = 0$ then no causal successor f of e can be executed in the same run, and so $x(f)$ must be equal to 0. These observations can be formalized by considering Σ^Θ-compatible vectors (see Section 1.4 for the definition), and the following result provides a basis for such an approach.

Proposition 7.4. *A vector $x \in \{0,1\}^q$ is Σ^Θ-compatible iff for all distinct events $e, f \in E$ such that $x(e) = 1$, we have:*

$$f \prec e \Rightarrow x(f) = 1 \quad \text{and} \quad f\#e \Rightarrow x(f) = 0 \; . \tag{7.9}$$

Note: Σ^Θ-compatible vectors are binary, since each event in the unfolding of Σ can occur at most once in an execution sequence.

Proof. ($\Rightarrow$) Let σ be an execution sequence starting from M_{in}, such that $x_\sigma = x$. For $e \in E$ to be executed, it is necessary for all $f \in E$ satisfying $f \prec e$ to occur before e. Moreover, if $f\#e$ then f cannot happen in the same execution sequence. Hence (7.9) holds.

($\Leftarrow$) We will show that for each vector $x \in \{0,1\}^q$ satisfying (7.9), it is possible to build an execution sequence σ whose Parikh vector $x_\sigma = x$.

As shown in [77,78], for acyclic nets the feasibility of the marking equation is a sufficient condition for a marking to be reachable. Moreover, the proof of this result presented in [77,78] implies that any solution of this equation corresponds to at least one execution sequence σ. Hence, if for a given vector $x \in \{0,1\}^q$, $M = M_{in} + \mathfrak{I}_{\Sigma^\Theta} \cdot x \geq \mathbf{0}$ then M is reachable and there is an execution sequence leading to M whose Parikh vector is x. Therefore, it is enough to show that for every $x \in \{0,1\}^q$ satisfying (7.9), $M_{in} + \mathfrak{I}_{\Sigma^\Theta} \cdot x \geq \mathbf{0}$, i.e., that the following holds:

$$M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \geq 0 \quad \text{for all } b \in B \ . \tag{7.10}$$

Since for all $b \in B$, $|{}^\bullet b| \leq 1$,

$$\sum_{f \in {}^\bullet b} x(f) = \begin{cases} 0 & \text{if } {}^\bullet b = \emptyset \\ x(f') & \text{if } {}^\bullet b = \{f'\} \ , \end{cases}$$

and in what follows we consider two cases:

Case 1: ${}^\bullet b = \emptyset$. Then b is an initial condition and so $M_{in}(b) = 1$. In this case, (7.10) has the form $\sum_{f \in b^\bullet} x(f) \leq 1$, and it holds due to the second part of (7.9) and the fact that all the events in $b^\bullet$ are in conflict.

Case 2: ${}^\bullet b = \{f'\}$ for some $f' \in E$. Then $M_{in}(b) = 0$, and so (7.10) has the form $\sum_{f \in b^\bullet} x(f) \leq x(f')$, and it holds by (7.9), since all the events in $b^\bullet$ are in conflict, and f' is a predecessor for all of them. □

Corollary 7.5. *For each reachable marking M of Σ, there exists an execution sequence of Σ^Θ leading to a marking representing M, whose Parikh vector x satisfies (7.9), and for every $e \in E_{cut}$, $x(e) = 0$.*

Proof. Since the prefix used to build Σ^Θ was complete, each reachable marking M of Σ is represented in Σ^Θ by a marking M' which can be reached from M_{in} through an execution sequence σ without cut-off events. Proposition 7.4 implies that the Parikh vector of σ satisfies (7.9). □

There exists a one-to-one correspondence between Σ^Θ-compatible vectors and configurations of the finite and complete prefix which was taken as the basis of Σ^Θ. In view of the last result, it is sufficient for a deadlock detection algorithm to check only Σ^Θ-compatible vectors whose components corresponding to cut-off events are equal to zero. This can be done by freezing all $x(e)$ such that $e \in E_{cut}$ at the beginning of the algorithm and constructing the *minimal* Σ^Θ*-compatible closure* (defined below) of the current vector in each step of the algorithm.

7.3.3 Compatible closures

A Σ^Θ-compatible vector $y \in \{0,1\}^q$ is a Σ^Θ*-compatible closure* of a vector $x \in \{0,1\}^q$ if $x \leq y$. Moreover, y is the *minimal* Σ^Θ-compatible closure of x,

denoted by $MCC(x)$, if it is minimal with respect to $\leq$ among all possible Σ^Θ-compatible closures of x. Note that $MCC(x)$ can be undefined for some x's, but it is unambiguous whenever it is defined, due to Proposition 7.6 below.

As an example, let us consider the causal ordering $e_1 \prec e_2 \prec e_3$, $e_2 \prec e_4$ and e_3 *co* e_4 (see Figure 7.4), and $x = (1, 0, 1, 0)$. Then $y = (1, 1, 1, 0)$ and $z = (1, 1, 1, 1)$ are Σ^Θ-compatible closures of x, and $MCC(x) = y$.

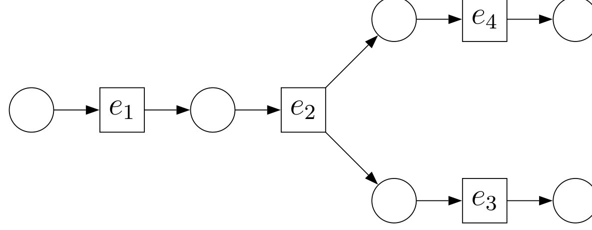

Figure 7.4: An occurrence net.

Proposition 7.6. *A vector $x \in \{0, 1\}^q$ has a Σ^Θ-compatible closure iff for all $e, f \in E$, $x(e) = x(f) = 1$ implies $\neg(e\#f)$. If x has a Σ^Θ-compatible closure then its minimal Σ^Θ-compatible closure exists and is unique. Moreover, in such a case if x has zero components for all cut-off events, then the same is true for $MCC(x)$.*

Proof. Straightforward. We just point out that in order to build the minimal Σ^Θ-compatible closure of x, when it does exist, it is enough to set to 1 all the components $x(f)$ for which there is e such that $f \prec e$ and $x(e) = 1$, i.e., to 'downclose' the set of events corresponding to x, producing a configuration. $\square$

From the implementation point of view, it may happen that a vector x has a Σ^Θ-compatible closure according to Proposition 7.6, but it cannot be computed because some of the zero components of x to be set to 1 have been frozen during the search process (see Section 7.2). In such a case, the algorithm should behave as if such a closure cannot be built.

7.3.4 Removal of redundant constraints

One can see that the inequalities in the middle of (7.8) are not essential for an algorithm checking only Σ^Θ-compatible vectors. Indeed, they are just the result of the substitution of $M = M_{in} + \mathfrak{I}_{\Sigma^\Theta} \cdot x$ into the constraint $M \geq \mathbf{0}$ and hold for any Σ^Θ-compatible vector x (see the proof of Proposition 7.4). Consequently, these inequalities may be left out without adding any Σ^Θ-compatible solution.

This gives the following reduced system of constraints:

$$\begin{cases} \sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) \text{ for all } e \in E \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \text{ is } \Sigma^\Theta\text{-compatible} . \end{cases} \tag{7.11}$$

7.3.5 Extending CDA (intuition)

Each step of *CDA* can be seen as moving from a point $\mathcal{A} \cdot x$ along a default vector $\mathcal{A} \cdot \varepsilon_j$ such that $\mathcal{A} \cdot x \odot \mathcal{A} \cdot \varepsilon_j < 0$, which is interpreted as 'returning to the origin' (see Figure 7.1). However, for an algorithm checking Σ^Θ-compatible vectors only, each step consists in moving along a vector which may be represented as a sum of *several* default vectors, and this branching condition is no longer valid. Indeed, let us consider the same ordering as in Figure 7.4, and the equation

$$\mathcal{A} \cdot x \stackrel{\text{df}}{=} x_1 + 5x_2 - 3x_3 - 3x_4 = 0$$

(which has a solution $x = (1,1,1,1)$) with an initial constraint $x_1 = 1$. Then the algorithm starts from the vector $x = (1,0,0,0)$, and the sequence of steps should begin from either ε_2 or $\varepsilon_2 + \varepsilon_3$ or $\varepsilon_2 + \varepsilon_4$. But $(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_2) = 5 \not< 0$, $(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot (\varepsilon_2 + \varepsilon_3)) = 2 \not< 0$, and $(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot (\varepsilon_2 + \varepsilon_4)) = 2 \not< 0$, so we cannot choose a vector to make the first step! A possible solution is to interpret each step $\varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ as a sequence of smaller steps $\varepsilon_{i_1}, \ldots, \varepsilon_{i_k}$ where we choose only the first element ε_{i_1} for which $\mathcal{A} \cdot \varepsilon_{i_1}$ does return to the origin, and then build the minimal Σ^Θ-compatible closure $x + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ of $x + \varepsilon_{i_1}$ without worrying where the vector $\varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ actually leads (if there is no Σ^Θ-compatible closure of $x + \varepsilon_{i_1}$ then ε_{i_1} cannot be chosen). This means that we check the condition $(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_{i_1}) < 0$ which coincides with the original *CDA*'s branching condition, though we are moving along a possibly different vector. The geometric interpretation of the new branching condition is shown in Figure 7.5. We will now cast the above idea in a formal setting.

7.3.6 Developing an extension of CDA

In this section, we will obtain a general result extending that in [18]. Consider the following homogeneous system of linear constraints:

$$\begin{cases} \mathcal{A} \cdot x = \mathbf{0} \\ \mathcal{B} \cdot x \leq \mathbf{0} \\ x \in D \stackrel{\text{df}}{=} D_1 \times \cdots \times D_q , \end{cases} \tag{7.12}$$

where $D_i \stackrel{\text{df}}{=} \{k_i, k_i + 1, \ldots, k_i + l_i\}$ and $k_i, l_i \geq 0$, for every $i \leq q$. Below we assume that $\mathbf{0} \notin D$.[4]

[4]From the point of view of this chapter, such an assumption is unproblematic. The case $\mathbf{0} \in D$ is discussed in Remark 7.12, at the end of this section.

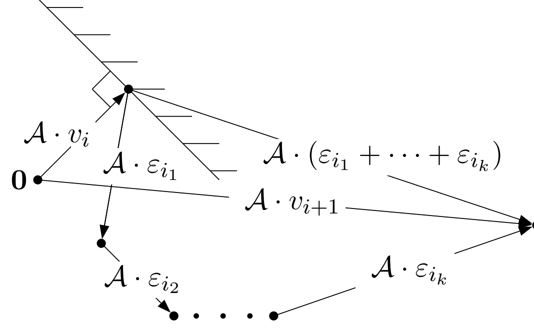

Figure 7.5: Geometric interpretation of the new branching condition: $\mathcal{A} \cdot \varepsilon_{i_1}$ is 'returning to the origin', although $\mathcal{A} \cdot (\varepsilon_{i_1} + \cdots + \varepsilon_{i_k})$ does not necessarily posses this property; here $v_{i+1} = v_i + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ is the minimal Σ^{Θ}-compatible closure of $v_i + \varepsilon_{i_1}$.

Let $\xi : D \to D$ be a partial function[5] with the domain dom_ξ such that $x_{min} \stackrel{\text{df}}{=} (k_1, \ldots, k_q) \in dom_\xi$, and $codom_\xi \stackrel{\text{df}}{=} \xi(dom_\xi)$. A *$\xi$-minimal solution* of (7.12) is any solution $x \in codom_\xi$ for which there is no solution $y \in codom_\xi$ satisfying $y < x$. We will denote this by $x \in min_\xi$, and assume that:

$$\begin{aligned} y \in dom_\xi &\implies y \le \xi(y) \\ y \le x \in min_\xi &\implies y \in dom_\xi \ \wedge \ \xi(y) \le x \ . \end{aligned} \tag{7.13}$$

The aim is to develop an algorithm enumerating the ξ-minimal solutions and, in what follows, we present an extension of *CDA* achieving this. First, we introduce a new branching condition.

Branching Condition 3. *A vector $x \in codom_\xi$ which is not a solution of (7.12) can be extended to $\xi(x + \varepsilon_j)$ if $x + \varepsilon_j \in dom_\xi$ and*

$$(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot x\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} < 0 \ , \tag{7.14}$$

where m is the number of rows in $\mathcal{B}$, and $\mathcal{B}_i$ is the i-th row of $\mathcal{B}$. ◇

The above rule determines a search space which can be represented by a labelled directed graph $G_\xi \stackrel{\text{df}}{=} (X, A)$, where $X \subseteq codom_\xi$ is a set of vertices and $A \subseteq X \times \mathcal{CB} \times X$ is a set of arcs. It is defined as the smallest graph such that X contains a distinguished vertex $x_{root} \stackrel{\text{df}}{=} \xi(x_{min})$ and, for every $x \in X$ which is not a solution of (7.12), if $\varepsilon_j \in \mathcal{CB}$ satisfies $x + \varepsilon_j \in dom_\xi$ and (7.14), then $(x, \varepsilon_j, \xi(x + \varepsilon_j)) \in A$. Directly from the definitions we obtain

Proposition 7.7. *G_ξ is finite and acyclic.*

[5]In Section 7.3.7 we will take ξ to be the *MCC* function.

Proof. From the first part of (7.13) it follows that $y < x$, for every $(y, \varepsilon_j, x) \in A$. Thus a directed path in G_ξ can have at most $|D|$ vertices. The result follows from this and $|X| \leq |D| < \infty$. □

The next proposition states a crucial property of the new branching condition.

Proposition 7.8. *If a vertex y of G_ξ and $x \in min_\xi$ are satisfying $y < x$, then there is an arc (y, ε_j, z) in G_ξ such that $z \leq x$.*

Proof. (Adapted from [2].) We have $x - y = \sum_{j\in\mathcal{J}} \varepsilon_j$, for some non-empty multiset $\mathcal{J}$. Suppose that the desired arc does not exist. We observe that, for every $j \in \mathcal{J}$, by the second part of (7.13), $y + \varepsilon_j \in dom_\xi$ and $\xi(y + \varepsilon_j) \leq x$. Thus, for all $j \in \mathcal{J}$

$$(\mathcal{A} \cdot y) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot y\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} \geq 0 \ ,$$

and after summing these inequalities for all $j \in \mathcal{J}$, we obtain

$$\begin{array}{l} (\mathcal{A} \cdot y) \odot (\mathcal{A} \cdot (x - y)) \\ \quad + \displaystyle\sum_{j\in\mathcal{J}} \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot y\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} \geq 0 \ . \end{array} \tag{7.15}$$

Let $\mathcal{K}_{>0}$ and $\mathcal{K}_{\leq 0}$ be the sets of all $i \leq m$ such that $\mathcal{B}_i \odot y > 0$ and $\mathcal{B}_i \odot y \leq 0$, respectively. Since $\mathcal{A} \cdot x = \mathbf{0}$,

$$\begin{array}{l} \displaystyle\sum_{j\in\mathcal{J}} \sum_{i\in\mathcal{K}_{>0}} (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j) \\ \quad \geq ||\mathcal{A} \cdot y||^2 - \displaystyle\sum_{j\in\mathcal{J}} \sum_{i\in\mathcal{K}_{\leq 0}} \min\{(\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j), 0\} \geq 0 \ . \end{array}$$

We are now going to show that $\mathcal{K}_{>0} = \emptyset$. Indeed, by the last inequality,

$$\sum_{j\in\mathcal{J}} \sum_{i\in\mathcal{K}_{>0}} (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j) = \sum_{i\in\mathcal{K}_{>0}} (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot (x - y)) \geq 0 \ .$$

This, and the fact that for all $i \in \mathcal{K}_{>0}$, $(\mathcal{B}_i \odot y)(\mathcal{B}_i \odot x) \leq 0$ (which follows from $\mathcal{B} \cdot x \leq \mathbf{0}$ and the definition of $\mathcal{K}_{>0}$), yields

$$0 \geq \sum_{i\in\mathcal{K}_{>0}} (\mathcal{B}_i \odot y)(\mathcal{B}_i \odot x) \geq \sum_{i\in\mathcal{K}_{>0}} (\mathcal{B}_i \odot y)^2 \geq 0 \ .$$

Hence $\mathcal{B}_i \odot y = 0$, for all $i \in \mathcal{K}_{>0}$. This, however, means that $\mathcal{K}_{>0} = \emptyset$. As a result, $\mathcal{B} \cdot y \leq \mathbf{0}$.

From $\mathcal{K}_{>0} = \emptyset$ it further follows that $\max\{\mathcal{B}_i \odot y, 0\} = 0$, for all $i \leq m$, which together with (7.15) and $\mathcal{A} \cdot x = \mathbf{0}$ leads to

$$\sum_{j\in\mathcal{J}} \sum_{i=1}^{m} \min\{(\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j), 0\} \geq ||\mathcal{A} \cdot y||^2 \geq 0 \ .$$

Thus, since $\min\{(\mathcal{B}_i \odot y)(\mathcal{B}_i \odot \varepsilon_j), 0\} \leq 0$, for every $i \leq m$, we obtain that $\mathcal{A}\cdot y = \mathbf{0}$. Hence $y \in codom_\xi$ is a solution of (7.12) satisfying $y < x$, contradicting $x \in min_\xi$. □

Corollary 7.9. *All ξ-minimal solutions are vertices of G_ξ.*

Proof. Let $x \in min_\xi$. We first observe that $x_{root} \leq x$ which follows from the second part of (7.13). Hence $x \in X$, by Propositions 7.7 and 7.8. □

Although the above corollary and Proposition 7.7 imply that G_ξ could be used to solve the problem at hand,[6] it may contain a large number of redundant paths. We will now adapt the frozen components method of [2, 18] to cope with this problem. Below, for any node x of G_ξ we denote by $out(x)$ the set of all the ε_j's which label the arcs outgoing from x.

Frozen Components 2. *We assume that, for each node x of G_ξ, there is a total ordering $\prec_x$ on the set $out(x)$. And, if $\varepsilon_i \prec_x \varepsilon_j$, then ε_j is* frozen *along all the directed paths in G_ξ beginning with the arc $(x, \varepsilon_i, \xi(x + \varepsilon_i))$.* ◇

To capture the above rule through a suitable modification of G_ξ, we associate sets of frozen components with the arcs of directed paths originating at x_{root}. Let $\sigma = a_1 a_2 \ldots a_k$ be a sequence of arcs in G_ξ forming a directed path starting at x_{root}. For every arc $a_i = (x, \varepsilon_j, y)$ in σ, we denote by $Froz_\sigma(a_i)$ a subset of $\mathcal{CB}$ such that

$$Froz_\sigma(a_i) \stackrel{\text{df}}{=} \{\varepsilon_m \in out(x) \mid \varepsilon_j \prec_x \varepsilon_m\} \cup \begin{cases} \emptyset & \text{if } i = 1 \\ Froz_\sigma(a_{i-1}) & \text{if } i > 1\ . \end{cases}$$

We then say that σ is *non-frozen* if, for every arc $a_i = (x, \varepsilon_j, y)$ in σ, $Supp(y - x) \cap Froz_\sigma(a_i) = \emptyset$, where $Supp(x) \stackrel{\text{df}}{=} \{\varepsilon_j \mid \varepsilon_j \leq x\}$.

With the above notation, Frozen Components 2 determines a search space which can be represented by the smallest subgraph T_ξ of G_ξ containing x_{root} and all the non-frozen directed paths of G_ξ.

Proposition 7.10. *T_ξ is a tree rooted at x_{root} whose set of vertices contains all ξ-minimal solutions.*

Proof. We first observe that the orderings associated with the vertices of G_ξ induce, for every vertex x, a total order $\ll_x$ on all the directed paths leading from x_{root} to x in such a way that, $\sigma \ll_x \sigma'$ iff $\sigma = \sigma_1(y, \varepsilon_i, z)\sigma_2$, $\sigma' = \sigma_1(y, \varepsilon_j, z')\sigma_3$ and $\varepsilon_i \prec_x \varepsilon_j$ (note that since G_ξ is acyclic, a directed path leading from x_{root} to x cannot be a prefix of another directed path from x_{root} to x).

Suppose that T_ξ is not a tree. Then there are two different non-frozen directed paths, $\sigma \ll_x \sigma'$, leading from x_{root} to some node $x \neq x_{root}$. We can represent them as $\sigma = \sigma_1(y, \varepsilon_i, z)\sigma_2$ and $\sigma' = \sigma_1(y, \varepsilon_j, z')\sigma_3$, where $\varepsilon_i \neq \varepsilon_j$. Then $\varepsilon_j \in Froz_\sigma(a)$,

[6] For example, G_ξ could be searched in the breadth-first or depth-first manner.

for every arc in $(y, \varepsilon_i, z)\sigma_2$. Moreover, by the first part of (7.13), $\varepsilon_j \leq x - y$ and so σ is not non-frozen, a contradiction.

Suppose now that $x \in min_\xi$. Since G_ξ is finite, and there is at least one directed path from x_{root} to x, there is a unique directed path $\sigma = a_1 \dots a_k$ from x_{root} to x which is maximal w.r.t. $\ll_x$. Suppose that such a σ is not non-frozen. Then there are $m \in \{1, \dots, k\}$ and $a \in A$ such that $a_m = (y, \varepsilon_j, \xi(y+\varepsilon_j))$, $a = (y, \varepsilon_i, \xi(y+\varepsilon_i))$, $\varepsilon_j \prec_y \varepsilon_i$ and $\varepsilon_i \in Supp(x-y)$. By the second part of (7.13), $\xi(y + \varepsilon_i) \leq x$. Hence, by Propositions 7.7 and 7.8, there is a directed path $\sigma' = aa'_1 \dots a'_l$ from $\xi(y + \varepsilon_i)$ to x. Thus $\sigma'' = a_1 \dots a_{m-1}\sigma'$ is a directed path in G_ξ such that $\sigma \ll_x \sigma''$, contradicting the choice of σ. Hence x is a vertex of T_ξ. □

We observe that since T_ξ is a tree, in the notation $Froz_\sigma(a)$ we can drop the index σ (see the definition of $Froz_\sigma$).

The above frozen components rule allows for further improvement, which can be given in the form of an additional function *froz*.

Frozen Components 3. *We assume that, for every arc a of T_ξ, $froz(a)$ is a subset of $\mathcal{CB}$ such that if a and a' form two consecutive arcs then $froz(a) \subseteq froz(a')$. Moreover, if $a_1 \dots a_k$ is a directed path in T_ξ leading from x_{root} to $y \in min_\xi$, then for every $i \leq k$, $froz(a_i) \cap Supp(y - x_i) = \emptyset$, where x_i is the origin of a_i.* ◇

Proposition 7.11. *Let S_ξ be the minimal subtree of T_ξ which contains x_{root} and all the directed paths non-frozen w.r.t. froz. Then the set of vertices of S_ξ comprises all ξ-minimal solutions.*

Proof. Follows directly from the definitions and (7.13). □

To summarize, Branching Condition 3 and Frozen Components 2 and 3 define the search tree which can be traversed[7] to find all ξ-minimal solutions of (7.12) in a finite number of steps (as G_ξ is finite, see Proposition 7.7).

The resulting approach can then be applied to deal with a non-homogeneous system of linear constraints

$$\begin{cases} \mathcal{A} \cdot x = \alpha \\ \mathcal{B} \cdot x \leq \beta \\ x \in D \end{cases} \tag{7.16}$$

where we *do not* assume that $\mathbf{0} \notin D$, and all the notions and assumptions relating to ξ are as those for (7.12).

The problem of finding all ξ-minimal solutions of (7.16) can be reduced to an instance of the problem considered earlier in this section. To this end, we

[7] Using the depth-first search as the breadth-first search would be inefficient due to the need of recording frozen components.

introduce an auxiliary variable z and two matrices, $\mathcal{A}' \stackrel{\mathrm{df}}{=} (\mathcal{A}, -\alpha)$ and $\mathcal{B}' \stackrel{\mathrm{df}}{=} (\mathcal{B}, -\beta)$. Then (7.16) can be rewritten as

$$\begin{cases} \mathcal{A}' \cdot (x, z) = \mathbf{0} \\ \mathcal{B}' \cdot (x, z) \leq \mathbf{0} \\ (x, z) \in D' \stackrel{\mathrm{df}}{=} D \times \{1\} \; . \end{cases} \tag{7.17}$$

Moreover, after setting $dom'_\xi \stackrel{\mathrm{df}}{=} dom_\xi \times \{1\}$ and $\xi'(x, z) \stackrel{\mathrm{df}}{=} (\xi(x), 1)$, we obtain an instance of (7.12) (note that $\mathbf{0} \notin D'$). We now observe that x is a ξ-minimal solution of (7.16) iff $(x, 1)$ is a ξ'-minimal solution of (7.17). As a result, we can render the branching condition derived for (7.17), directly in terms of (7.16).

Branching Condition 4. *A vector $x \in codom_\xi$ which is not a solution of (7.16) can be extended to $\xi(x + \varepsilon_j)$ if $x + \varepsilon_j \in dom_\xi$ and*

$$(\mathcal{A} \cdot x - \alpha) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} r_i < 0 \; , \tag{7.18}$$

where, for every $i \in \{1, \ldots, m\}$,

$$r_i \stackrel{\mathrm{df}}{=} \begin{cases} 0 & \text{if } \mathcal{B}_i \odot x \leq \beta_i \wedge \mathcal{B}_i \odot \varepsilon_j \leq 0 \\ (\mathcal{B}_i \odot x - \beta_i)(\mathcal{B}_i \odot \varepsilon_j) & \text{otherwise} \; . \end{cases}$$

◇

Remark 7.12. We assume that $x_{min} \in dom_\xi$ since otherwise there are no ξ-minimal solutions at all.

To obtain a full extension of *CDA*, we still need to consider (7.12) when $\mathbf{0} \in D$ (note that $\mathbf{0}$ is a trivial solution and has to be excluded from the search). The discussion can easily be adapted, as follows:

- We assume that $\mathbf{0} \notin codom_\xi$.
- $x_{root} \stackrel{\mathrm{df}}{=} \mathbf{0}$, and if $\varepsilon_j \in dom_\xi$ then $(\mathbf{0}, \varepsilon_j, \xi(\varepsilon_j)) \in A$.

Then all the results developed earlier in this section still hold, in particular, Propositions 7.10 and 7.11.

Allowing infinite ranges $D_i \stackrel{\mathrm{df}}{=} \{k_i, k_i + 1, \ldots\}$ leads to termination problems; in other words, the search graph G_ξ may be infinite. In such a case, one needs to develop conditions for bounding ξ-minimal solutions. Such a problem depends on the actual definition of the function ξ, and so we expect that it will be addressed on an individual basis. Here, we assume that the domain is finite and so the termination is always guaranteed. ◇

7.3.7 Applying the method for Σ^Θ-compatible vectors

We will now apply the theory developed in the previous section to check only Σ^Θ-compatible vectors. This can be done due to the fact that the *MCC* function satisfies (7.13). Indeed, if $y \in dom_{MCC}$ then $y \leq MCC(y)$ by the definition of a Σ^Θ-compatible closure; moreover, if $y \leq x \in min_{MCC}$ then x is one of the compatible closures of y, and thus $y \in dom_{MCC}$ by Proposition 7.6, and $MCC(y) \leq x$ by the definition of the minimal Σ^Θ-compatible closure.

Referring to the notation introduced above, we shall assume that the system of constraints to be solved is a non-homogeneous one, and:

- $D_i \overset{\text{df}}{=} \{0\}$ if $e_i \in E_{cut}$, and $D_i = \{0, 1\}$ otherwise.
- dom_ξ is the set of all vectors of D having a Σ^Θ-compatible closure, and $\xi(x) \overset{\text{df}}{=} MCC(x)$.
- For an arc $a = (x, \varepsilon_j, y)$, $froz(a) \overset{\text{df}}{=} \{\varepsilon_i \mid \exists \varepsilon_k \in Supp(y) : e_k \# e_i\}$.

It is straightforward to show that all the properties required for dom_ξ, ξ and *froz* are then satisfied, and so after ignoring the auxiliary variable z, the search tree S_ξ contains all minimal Σ^Θ-compatible solutions.

7.4 Implementation of the algorithm

The idea of the algorithm in Figure 7.6 is very similar to that of CDA modified to solve non-homogeneous systems of linear constraints and checking only Σ^Θ-compatible vectors, such that $x(e) = 0$ for all $e \in E_{cut}$.

In general case, the maximal depth of the search tree is $O(q)$ (recall that q is the number of events in the prefix), so CDA would need to store $O(q)$ vectors of length q. The proposed algorithm uses just two arrays, X and *Fixed*, both of length q:

X : *array*[*1..q*] *of* $\{0, 1\}$ To construct a solution.

Fixed : *array*[*1..q*] *of integers* To keep the information about the levels of fixing the components of X.

The interpretation of these arrays is as follows:

Fixed[*i*] $= 0$ Then $X[i]$ must be 0 and this means that $X[i]$ has not yet been considered, and may later be set to 1 or frozen.

Fixed[*i*] $= k > 0 \wedge X[i] = 0$ Then $X[i]$ has been frozen at some node on level k whose subtree the algorithm is developing. It cannot be unfrozen until the algorithm backtracks to level k.

input :
  $Cons$ — a system of constraints
  Σ^Θ — a net system built from Unf_Σ^Θ
output :
  X — a Σ^Θ-compatible solution of $Cons$, if there is one
initialization
  $depth \leftarrow 1$
  $X \leftarrow (0, \ldots, 0)$
  for $i \in \{1, \ldots, q\}$: $Fixed[i] \leftarrow \begin{cases} 1 & \text{if } e_i \in E_{cut} \\ 0 & \text{otherwise} \end{cases}$
procedure *solve*
  if X is a solution of $Cons$ **then stop**
  for all $i \in \{1, \ldots, q\}$ **such that** (7.18) holds **do**
    if $Fixed[i] = 0$ **then**
      $depth \leftarrow depth + 1$
      $freeze(i)$
      $solve$
      $clear$
      $depth \leftarrow depth - 1$
      $fix_vars(i)$

Figure 7.6: Integer programming verification algorithm.

$Fixed[i] = k > 0 \wedge X[i] = 1$ Then $X[i]$ has been set to 1 at some node on level k whose subtree the algorithm is developing. This value is fixed for the entire subtree.

Notice that storing the levels of fixing the elements of X allows one to undo changes during backtracking, without keeping all the intermediate values of X. We also use the following auxiliary variables and procedures (see Figure 7.7):

$depth : integer$ The current depth in the search tree.

$freeze(i : integer)$ Freezes all $X[k]$'s such that $e_i \preceq e_k$. The corresponding elements of *Fixed* are set to the current value of *depth*.

$fix_vars(i : integer)$ Sets all $X[j]$'s such that $e_j \preceq e_i$ to 1 and uses *freeze* to freeze all $X[k]$'s such that $e_i \# e_k$. The current value of *depth* is written in the elements of *Fixed*, corresponding to the components being fixed.

clear Resets all $X[j]$'s and $Fixed[j]$'s which have been fixed on depth *depth*.

procedure *freeze*$(i : integer)$
  if $Fixed[i] = 0$ **then**
    $Fixed[i] \leftarrow depth$
    for all $e_k \in (e_i^\bullet)^\bullet$ **do** *freeze*(k)

procedure *fix_vars*$(i : integer)$
  if $Fixed[i] = 0$ **then**
    $X[i] \leftarrow 1$
    $Fixed[i] \leftarrow depth$
    for all $b \in {}^\bullet e_i$ **do**
      for all $e_k \in {}^\bullet b$ **do** *fix_vars*(k) /* $|{}^\bullet b| \leq 1$ */
      for all $e_k \in b^\bullet \setminus \{e_i\}$ **do** *freeze*(k)

procedure *clear*
  for all $i \in \{1 \dots q\}$ **such that** $Fixed[i] = depth$ **do**
    $X[i] \leftarrow 0$
    $Fixed[i] \leftarrow 0$

Figure 7.7: Auxiliary functions.

7.4.1 Retrieving a solution

What we often want to see as a solution is an execution sequence of the original net system, rather than a configuration of its unfolding. To derive such a sequence, it is enough to topologically sort the constructed configuration according to the causal order on the set of the events, and replace the events by their labels in the constructed sequence. An observation one can make is that the unfolders add events one-by-one to the unfolding being constructed, in such a way that for all non-cut-off events e_i and e_j, $e_i \prec e_j$ implies $i < j$. Therefore, we can avoid sorting the events and find a sequence of transitions in a straightforward way if the natural numbering of the components of the vector x, $x_i \stackrel{\text{df}}{=} x(e_i)$, is used.

7.4.2 Shortest trail

Finding a shortest path leading, e.g., to a deadlock can facilitate the debugging of a reactive system modelled by a Petri net. In such a case, we need to solve an optimization problem with the same system of constraints, and $\|x\| = x_1 + \cdots + x_q$ as the cost function to be minimized.

The algorithm can easily be adopted for this task. We do not stop after the first solution has been found, but keep the current optimal solution together with the corresponding value of the function $\|\cdot\|$. As this function is non-decreasing, we may prune a branch of the search tree as soon as the value of $\|\cdot\|$ becomes

greater than, or equal to, the current optimal value. This strategy speeds up the search and saves us from keeping all ξ-minimal solutions found so far. It is easy to see that this strategy does not affect the completeness, in the sense that a ξ-minimal solution minimizing $\|\cdot\|$ will always be found by the algorithm. Indeed, the strategy builds the same search tree up to the cutting of some of the subtrees rooted in nodes with the sum of the components not less than the optimal value of $\|\cdot\|$. But all the descendants of such nodes have even greater sum of the components, and so these subtrees cannot contain an optimal solution. To allow more pruning and, therefore, to reduce the search space, it makes sense to organize the search process in such a way that the first found solutions give the value of $\|\cdot\|$ 'close' to the optimal one. This can be done by choosing in each step of the algorithm the 'most promising' branches. Since the orderings $\prec_x$ used by the algorithm are arbitrary, we may exploit the information about the value of $\|\cdot\|$ on them, and check successors with smaller values first (see, e.g., the $\prec_{\|\cdot\|}$ ordering in [18]). Such an algorithm can be seen as a version of the 'branch and bound' method which considers only Σ^Θ-compatible vectors and uses frozen components and branching condition to reduce the search space.

7.5 Optimizations

Various heuristics used by general purpose integer programming solvers can be implemented to reduce the search effort. For example, one can look one step ahead and choose the branch which is in some sense the 'most promising' one. This can be done by choosing an ordering on the sons of each node of the search tree, depending on the current value of x (e.g., the $\prec_{\|\cdot\|}$ ordering in [18]).

Moreover, if the algorithm, having *fixed* some of the variables,[8] finds out that some of the inequalities have become infeasible, then it may prune the current branch of the search tree. Alternatively, it is sometimes possible to determine the values of some variables which have not yet been fixed, or to find out that some of the constraints have become redundant.

We now introduce some simple yet useful heuristics of this sort. Let us consider the inequality

$$\mathcal{B}_{i1}x_1 + \cdots + \mathcal{B}_{iq}x_q \leq \beta_i \tag{7.19}$$

in the context of generating the search tree at the current node and calculate

$$\mathit{fix} = \sum_{x_j \text{ is fixed}} \mathcal{B}_{ij}x_j \qquad \mathit{max} = \sum_{\substack{\mathcal{B}_{ij}>0 \\ x_j \text{ is not fixed}}} \mathcal{B}_{ij} \qquad \mathit{min} = \sum_{\substack{\mathcal{B}_{ij}<0 \\ x_j \text{ is not fixed}}} \mathcal{B}_{ij} \, .$$

Note that since $x \in \{0,1\}^q$, *min* and *max* are the bounds for the minimal and maximal possible values of the non-fixed part of the left hand side of the inequality in the subtree rooted in the current node, and one can show the following:

[8] x_i is fixed if it is equal to the highest value in D_i, or if ε_i has been frozen.

- If $min > \beta_i - \mathit{fix}$ then (7.19) (and so the whole system) is infeasible and the current subtree of the search tree may be pruned.
- If $max \leq \beta_i - \mathit{fix}$ then (7.19) is redundant and may be ignored in the subtree rooted at the current node of the search tree.
- If $min = \beta_i - \mathit{fix}$ then (7.19) can only be satisfied if all its non-fixed variables with negative coefficients are equal to 1, and all its non-fixed variables with positive coefficients are equal to 0. After fixing these variables (7.19) becomes redundant.
- If $min + \mathcal{B}_{ik} > \beta_i - \mathit{fix}$ for a non-fixed variable x_k (in this case $\mathcal{B}_{ik} > 0$), then (7.19) forces x_k to be 0.
- If $min - \mathcal{B}_{ik} > \beta_i - \mathit{fix}$ for a non-fixed variable x_k (in this case $\mathcal{B}_{ik} < 0$), then (7.19) forces x_k to be 1.

After fixing the value of a variable, it is necessary to build the minimal Σ^Θ-compatible closure of the current vector. As new variables can become fixed during this process, the above tests can be applied iteratively. If the minimal Σ^Θ-compatible closure cannot be built due to frozen components, then the current subtree of the search tree contains no Σ^Θ-compatible solution and may be pruned.

As an example, let us consider the following system of inequalities:

$$\begin{cases} -x_1 - 2x_2 + 2x_3 + 2x_4 + x_5 \leq 1 \\ x_1 + 2x_2 + x_4 - x_5 \leq 2 \\ 2x_1 + 3x_2 + x_3 - 3x_6 + 2x_7 + x_8 \leq 3 \,, \end{cases}$$

where the variables x_1 and x_2 are fixed to 1, and x_3 is fixed to 0. For the first inequality, $\mathit{fix} = -3$, $min = 0$, and $max = 3$, and so it is redundant due to $max \leq 1 - \mathit{fix}$. For the second inequality, $\mathit{fix} = 3$, $min = -1$, and $max = 1$. Due to $min = 2 - \mathit{fix}$, one can fix $x_4 = 0$ and $x_5 = 1$. For the third inequality, $\mathit{fix} = 5$, $min = -3$, and $max = 3$, and one can fix $x_6 = 1$ (due to $min - (-3) > 3 - \mathit{fix}$) and $x_7 = 0$ (due to $min + 2 > 3 - \mathit{fix}$). After fixing these variables, the inequality becomes redundant.

Suppose now that we added another constraint, $x_1 + x_2 - 2x_3 + x_9 \leq 1$, for which $\mathit{fix} = 2$, $min = 0$ and $max = 1$. Then the system becomes infeasible due to $min > 1 - \mathit{fix}$.

Such optimization rules can formally be justified in the following way. Let $opt : D \to D$ be a partial function[9] with the domain dom_{opt}, corresponding to applying the heuristics described above, satisfying:

$$\begin{aligned} x \in dom_{opt} &\Longrightarrow x \leq opt(x) \\ x \leq y \in min_\xi &\Longrightarrow x \in dom_{opt} \ \wedge \ opt(x) \leq y \,. \end{aligned} \tag{7.20}$$

[9]Intuitively, $opt(x)$ is undefined if, during the application of the optimization rules, the algorithm finds out that the system has no ξ-minimal solution $y \geq x$.

We then define a partial function $\xi_o : D \to D$ such that $\xi_o(x) \stackrel{\text{df}}{=} \xi(opt(\xi(x)))$, for every x in dom_{ξ_o} which is the largest subset of dom_ξ for which this expression is well-defined. We denote $codom_{\xi_o} \stackrel{\text{df}}{=} \xi_o(dom_{\xi_o})$, and then observe that, by (7.13) and (7.20):

$$\begin{array}{rcl} x \in dom_{\xi_o} & \Longrightarrow & x \le \xi_o(x) \\ x \le y \in min_{\xi_o} & \Longrightarrow & x \in dom_{\xi_o} \ \wedge \ \xi_o(x) \le y \ . \end{array} \tag{7.21}$$

Proposition 7.13. $min_\xi = min_{\xi_o}$.

Proof. Suppose that $x \in min_\xi$. Then, by $x \le x$ and (7.21), we have $x \le \xi_o(x) \le x$. Hence $\xi_o(x) = x$ and so $x \in codom_{\xi_o}$. If $x \notin min_{\xi_o}$, then there is $y \in min_{\xi_o}$ such that $y < x$. Hence, since $codom_{\xi_o} \subseteq codom_\xi$, we obtain a contradiction with $x \in min_\xi$.

Suppose that $y \in min_{\xi_o}$. If $y \notin min_\xi$ then, by $codom_{\xi_o} \subseteq codom_\xi$, there is $z \in min_\xi$ such that $z < y$. By the first part of the proof, $z \in min_{\xi_o}$, contradicting $y \in min_\xi$. ☐

From Proposition 7.13 and (7.21) it follows that the counterpart of (7.13) holds for ξ_o as well. Thus, in view of $min_\xi = min_{\xi_o}$, the search for ξ-minimal solutions can be based on the tree S_{ξ_o}, which can often be much more efficient than using S_ξ. As to the frozen components given by the function $froz_o$, it must satisfy Frozen Condition 3. We only note that, when solving (7.11), it is always possible to include into $froz_o(a)$ the set $\{\varepsilon_i \mid \exists \varepsilon_k \in Supp(y) : \ e_k \# e_i\}$, for every arc $a = (x, \varepsilon_j, y)$.

In order to avoid calculations related to redundant constraints, we can remember for each of them the depth in the search tree at which it was marked as redundant, and unmark it during the backtracking. Clearly, they do not need to be considered when checking whether the system is satisfied or performing the described optimizations.

7.6 Extended reachability analysis

The algorithm described in Section 7.4 is applicable to any system of linear constraints which are supposed to be solved in Σ^Θ-compatible vectors. The theory of verifying *co-linear* properties using unfoldings was developed in [76]. It can easily be generalized to arbitrary reachability properties, although solving (sometimes very large) non-linear systems obtained in this case is usually a hard task for general-purpose solvers. An algorithm checking only Σ^Θ-compatible vectors can do this more efficiently. Indeed, the only reason why the algorithm in Section 7.4 accepts only systems of linear constraints is that in order to reduce the search space it employs the branching condition (7.18). In principle, it can deal with arbitrary constraints, if one switches off this heuristic.

The approach we will now describe is similar to the one described in [76], generalized to deal with non-linear constraints. In addition, we use the ideas

from Section 7.3 to re-formulate the resulting integer programming problem in terms of Σ^Θ-compatible vectors.

Let us consider a property $Prop(M_1, \ldots, M_k)$ specified on markings of a net system Σ. We can transform it into a corresponding property $Prop'(x^1, \ldots, x^k)$ specified on Σ^Θ-compatible vectors $x^1, \ldots, x^k$ in such a way that if there exist markings $\hat{M}_1, \ldots, \hat{M}_k \in \mathcal{RM}(\Sigma)$ such that $Prop(\hat{M}_1, \ldots, \hat{M}_k)$ holds then $Prop'(\hat{x}^1, \ldots, \hat{x}^k)$ holds for some Σ^Θ-compatible vectors $\hat{x}^1, \ldots, \hat{x}^k$, and vice versa. Indeed, let $M \stackrel{\text{df}}{=} Mark(C_x)$ be the final marking of the configuration C_x given by a Σ^Θ-compatible vector x. Then $M' \stackrel{\text{df}}{=} Cut(C_x)$ is a corresponding marking of Σ^Θ. For every $p \in P$, $M(p)$ can be expressed as

$$M(p) = \sum_{h(b)=p} M'(b) \ ,$$

where $M'(b)$ can be found from the marking equation

$$M'(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \ .$$

Therefore,

$$M(p) = \sum_{h(b)=p} \left(M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \ . \tag{7.22}$$

Denoting by $Mark_p(x)$ the linear function in the right hand side of (7.22), we can express the final marking of the configuration C_x given by its Parikh-vector x as $Mark(x) \stackrel{\text{df}}{=} (Mark_{p_1}(x), \ldots, Mark_{p_m}(x))$, where $m = |P|$. Thus $Prop(M_1, \ldots, M_k)$ can be rendered as a predicate

$$Prop'(x^1, \ldots, x^k) \stackrel{\text{df}}{=} Prop(Mark(x^1), \ldots, Mark(x^k))$$

specified on Σ^Θ-compatible vectors. What is more, if *Prop* is initially expressed as a system of linear constraints then *Prop*$'$ will possess this property as well.

7.6.1 Deadlock checking in safe case

Applying the technique described in the previous section one can generate an alternative system of constraints for deadlock checking safe Petri nets (a similar idea was used in [76] to obtain a translation of this problem into a *MIP* problem). To begin with, we have the following condition for safe nets, stating that no transition is enabled:

$$\sum_{p \in {}^\bullet t} M(p) \leq |{}^\bullet t| - 1 \quad \text{for all } t \in T \setminus T_{dead} \ , \tag{7.23}$$

where T_{dead} is the set of transitions which are dead in Σ. Rendering it in terms of Σ^Θ-compatible vectors yields the following system of linear constraints (all

non-dead transitions needed for constructing this system can easily be found, as we have a finite and complete prefix):

$$\begin{cases} \sum_{p \in {}^\bullet t} Mark_p(x) \leq |{}^\bullet t| - 1 \quad \text{for all } t \in T \setminus T_{dead} \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \text{ is } \Sigma^\Theta\text{-compatible} . \end{cases} \tag{7.24}$$

In contrast to the method based on (7.11), one now has to keep in memory $|T \setminus T_{dead}|$ rather than $|E|$ constraints. Though the constraints are now longer, the overall size of the whole system (in terms of the number of monomials) is often much smaller (see Section 7.12).

Note that this method is in some sense more general than the one described in Section 7.1. In the latter, the cut-off events played an essential role in separating real deadlocks from the false ones, introduced by truncating the unfolding. But, in order to save some memory, one can omit cut-off events when generating a prefix. The approach proposed in this section, unlike the one based on the system (7.11), will work with such a 'stripped' prefix.

To apply this approach to non-safe net systems, one can use instead of (7.23) the following constraints:

$$\sum_{p \in {}^\bullet t} sg(M(p)) \leq |{}^\bullet t| - 1 \quad \text{for all } t \in T \setminus T_{dead} \ ,$$

where $sg(0) = 0$ and $sg(n) = 1$ for every $n > 0$; or, alternatively,

$$\prod_{p \in {}^\bullet t} M(p) = 0 \quad \text{for all } t \text{ in } T \setminus T_{dead} \ .$$

Although the resulting system is non-linear, it can be dealt with by the algorithm in Section 7.4 with the branching condition (7.18) switched off.

7.6.2 Terminal markings

Some reactive systems can have states corresponding to a proper termination, which are considered to be different from deadlocks, even though they may enable no transitions. For example, the PEP tool (see, e.g., [6, 7]) works with a class of labelled nets, called *boxes* (see, e.g., [67]), which are essentially safe Petri nets with distinguished disjoint sets of *entry* and *exit* places, denoted by P_{in} and P_{out} respectively. The proper terminal marking of a box is defined as one that puts a token on each of the exit places and no tokens elsewhere. Such a false deadlock can be eliminated from the set of solutions by adding a new constraint, which holds for all but the terminal marking. As the relevant property $Prop(M)$ one can take

$$\sum_{p \in P_{out}} M(p) - \sum_{p \in P \setminus P_{out}} M(p) \leq |P_{out}| - 1$$

and, using the approach described earlier in this section, render this constraint in terms of Σ^{Θ}-compatible vectors and add it to (7.11) or (7.24).

One could slightly relax the notion of a terminal marking of a box, allowing dead tokens on internal (i.e., different from the entry and exit) places. Such a situation can be handled in a similar way using the constraint

$$\sum_{p\in P_{out}} M(p) - \sum_{p\in P_{in}} M(p) \leq |P_{out}| - 1 \ .$$

7.7 Other verification problems

In this section we consider checking mutual exclusion of places, and marking reachability and coverability problems. Since all these properties are either linear or co-linear, they can be verified using the approach proposed in [76]. We refine the technique proposed there by reducing the problems to purely integer ones and checking only Σ^{Θ}-compatible vectors.

7.7.1 Mutual exclusion

Suppose, one has to check whether two places, p and p', of Σ are mutually exclusive. This is the case if, for any $M \in \mathcal{RM}(\Sigma)$, at least one of them is empty, or, in other words, $M(p) \geq 1$ and $M(p') \geq 1$ cannot hold simultaneously. Using the technique described earlier in this section, one can state that a necessary and sufficient condition for p and p' to be mutually exclusive is the infeasibility of the following system of linear constraints:

$$\begin{cases} Mark_p(x) \geq 1 \\ Mark_{p'}(x) \geq 1 \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \text{ is } \Sigma^{\Theta}\text{-compatible} \ . \end{cases}$$

In the safe case, one can check the pairwise mutual exclusion of more than two places simultaneously, and still remain within the domain of linear constraints. Indeed, let $P_{ME} \subseteq P$ be a set of places whose mutual exclusion is to be checked. Then $\sum_{p\in P_{ME}} M(p) \geq 2$ must not hold for any $M \in \mathcal{RM}(\Sigma)$, and so the corresponding necessary and sufficient condition is the infeasibility of the following system:

$$\begin{cases} \displaystyle\sum_{p\in P_{ME}} Mark_p(x) \geq 2 \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \text{ is } \Sigma^{\Theta}\text{-compatible} \ . \end{cases} \qquad (7.25)$$

7.7.2 Reachability and coverability

Since it is trivial to express the standard reachability and coverability problems in terms of extended reachability, we give the translation directly. A marking M of Σ is reachable (coverable) iff the following system of linear constraints is feasible:

$$\begin{cases} Mark_p(x) \overset{(\geq)}{=} M(p) \quad \text{for all } p \in P \\ x(e) = 0 \quad \text{for all } e \in E_{cut} \\ x \in \{0,1\}^q \text{ is } \Sigma^\Theta\text{-compatible} . \end{cases}$$

This system can be simplified as follows: for the coverability problem one can leave out the constraints for which $M(p) = 0$ as they always hold, and for the reachability problem one can replace all such constraints by their sum. In the safe case, M should be a safe marking, i.e., for all $p \in P$, $M(p) \in \{0, 1\}$ (otherwise it is neither reachable, nor coverable), and further simplifications are possible. In particular, one can replace the constraints for which $M(p) = 1$ by their sum, reducing thus the system to a single constraint in the case of the coverability problem, and to a system of two constraints for the reachability problem. What is more, it is possible to replace $\geq$ by $=$ when checking coverability.

7.8 Further optimization for deadlock detection

The deadlock detection problem (7.11), has a very special structure, which can be further exploited. In particular, the maximal value of the left hand side of the inequality

$$\sum_{b \in {}^\bullet e} \left(M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1$$

is $|{}^\bullet e|$, even if we allow x to be non-Σ^Θ-compatible. Therefore, the i-th inequality in (7.11) can be falsified iff all the variables from Pos_i are equal to 1, and all the variables from Neg_i are equal to 0, where by Pos_i and Neg_i we denote the sets of the variables with respectively positive and negative coefficients. This means that we may mark the i-th inequality as redundant as soon as any of the variables from Pos_i becomes frozen at 0, or if any of the variables from Neg_i is set to 1. In addition to this simple redundancy test, one can apply on each step an infeasibility test for each non-redundant inequality of (7.11). Indeed, if for the inequality all the variables from Pos_i are set to 1, and all the variables from Neg_i are frozen at 0, then this inequality (and, thus, the whole system) cannot be satisfied, and the algorithm may stop developing the current branch of the search tree. Apart from this, if all but one variable from Pos_i are set to 1, and all the variables from Neg_i are frozen at 0, then the only way to prevent a contradiction is to freeze at 0 the remaining variable from Pos_i. And, similarly, if all the variables in Pos_i are set to 1, and all but one variable in Neg_i are frozen at 0,

then we may deterministically set the remaining variable to 1. In both cases, the constraint becomes redundant. Notice that these rules can be formally justified by choosing appropriate *opt* and $froz_o$ functions (see Section 7.4).

According to experiments, the above problem-specific heuristics turned out to be much more efficient than the general ones described in Section 7.5.

The safe case

Unfortunately, the above heuristics do not work for (7.24), where the inequalities are more complex. But one still can derive some problem-specific optimization rules.

The maximal value of the left-hand side of the inequality for a transition t on any Σ^{Θ}-compatible vector x is bounded by $|^{\bullet}t|$, since $Mark_p(x) \leq 1$ for safe net systems. Therefore, if for some inequality, say the i-th one, the value of its left-hand side is $|^{\bullet}t|$, then we can state the following:

- If all the variables from Neg_i are frozen at 0, then this inequality (and, thus, the whole system) can never be satisfied. Hence the algorithm may stop developing the current branch of the search tree.

- If all the variables in Pos_i are set to 1, and all but one variable in Neg_i are frozen, then we may deterministically set the remaining non-fixed variable to 1. After this the constraint becomes redundant.

Moreover, if the value of the left-hand side is $|^{\bullet}t| - 1$, and all the variables from Neg_i are frozen at 0, then the only way to prevent a contradiction is to freeze all the non-fixed variables from Pos_i. After this the constraint becomes redundant. Again, the correctness of these heuristics can be justified by choosing appropriate *opt* and $froz_o$ functions (see Section 7.4).

The problem-specific redundancy tests we obtained for (7.24) are relatively complex, and not as efficient as the test described above for (7.11). The reason is that the inequalities of (7.24) do not become redundant as often as those of (7.11), and we used the general min/max-tests developed in Section 7.5. In Section 7.12, we will discuss how this new method compares with other deadlock detection algorithms.

7.9 On-the-fly deadlock detection

Experimental results demonstrated that the algorithm outlined in Section 7.4 is usually fast, but the treated systems of constraints can be very large, even if the sparse matrix representation is used (see Table 7.4 in Section 7.12). This, in turn, can lead to page swapping when the prefix is large (see, e.g., the LAMP(4) example, where page swapping led to significant slowdown). Therefore, it is clearly desirable to find a way to reduce the memory demand, provided that this results in an increase of the running time only by a small factor.

One can notice that the structure of each constraint in (7.11) is rather simple, and it can be generated 'on-the-fly', at the moment it is needed. Indeed, the algorithm refers to the system of constraints when checking whether the system is satisfied, when computing the branching condition,[10] and when applying the optimization rules described in Section 7.8. All these can be efficiently done without explicitly generating the system of constraints, by exploring the sets ${}^\bullet({}^\bullet e)$ and $({}^\bullet e)^\bullet$ for all $e \in E$. An observation one can make here is that for any event $e \in E$, ${}^\bullet({}^\bullet e) \cap ({}^\bullet e)^\bullet = \emptyset$ (since $e' \in {}^\bullet({}^\bullet e) \cap ({}^\bullet e)^\bullet$ would mean that simultaneously $e' \prec e$, and either $e' = e$ or $e' \# e$). Therefore, the positive and negative coefficients for each constraint in (7.11) can be efficiently separated, and this can be exploited by the algorithm.

The LAMP(4) example shows that the effect of the described approach can be very significant for large prefixes: due to economical memory usage it avoided page swapping and as a result was faster by more than an order of magnitude.

The on-the-fly approach can, in fact, be applied to other verification problems considered in this chapter. However, the resulting gains would be less significant, as the corresponding systems of constraints are usually of moderate size.

7.10 Efficiency of the branching condition

As it was mentioned in Section 7.6, the branching condition can be switched off in order to make the algorithm applicable to non-linear systems of constraints. That is, the purpose of the branching condition is to speedup solving linear systems. Therefore, it is interesting to investigate how efficient this heuristic is.

The answer to this question turns out to be problem specific. Experimentally we found out that if one solves a deadlock detection problem using (7.11) or (7.24) then the branching condition (7.18) should be switched off as it has little or no effect on the choices made by the algorithm.

On the other hand, it is quite efficient for solving other model checking problems, in particular those mentioned in Section 7.7. Indeed, trying to speedup coverability analysis one might be tempted to use the following problem-specific branching condition:

Branching Condition 5. *Increment only those $x(e)$, for which the transition $h(e)$ produces a token in a place $p \in M$.* ◇

But in some cases the standard branching condition (7.18) is 'clever enough' to choose only such $x(e)$! Let us show that this indeed is the case for safe net systems. The constraints have the form

$$Mark_p(x) \geq 1 \quad \text{for all } p \in M \ ,$$

[10] As explained in Chapter 7.10, this is not strictly necessary, since the branching condition is better to be switched off when checking the deadlock freeness.

where M is the marking one wants to cover. They can be rewritten as

$$\left(\sum_{h(b)=p} M_{in}(b)\right) - Mark_p(x) \leq \left(\sum_{h(b)=p} M_{in}(b)\right) - 1 \quad \text{for all } p \in M \ .$$

If (7.18) holds for some ε_j then $r_i < 0$ for some i, i.e., $(\mathcal{B}_i \odot x - \beta_i)(\mathcal{B}_i \odot \varepsilon_j) < 0$. Since $Mark_p(x) \leq 1$ for safe nets, $\mathcal{B}_i \odot x \geq \beta_i$ always holds, and so $\mathcal{B}_i \odot \varepsilon_j < 0$, i.e., $\mathcal{B}_{ij}$ is negative. But this means that x_j is $x(e)$ for some event e such that $h(e)^\bullet \cap M \neq \emptyset$.

Note that the above argument can easily be modified for the cases when the constraints are added up and/or $\leq$ is replaced by $=$.

Moreover, experimentally we have found out that the branching condition is quite efficient for checking the mutual exclusion property. Table 7.1 presents the performance measurements for the LAMP(n) series of benchmarks (see Chapter 6) based on (7.25). The meaning of the columns in the table is as follows (from left to right): the name of the problem; the number of places which need to be checked for mutual exclusion; the number of explored compatible vectors and the time taken by the algorithm with the branching condition switched off; the number of explored compatible vectors and the time taken by the algorithm with the branching condition switched on. One can see that the speedups gained by using the branching condition can be quite substantial.

Problem		No BC		BC	
	$\lvert P_{ME}\rvert$	vec	time [s]	vec	time [s]
LAMP(2)	4	46	<0.01	16	<0.01
LAMP(3)	6	2260	0.95	505	0.78
LAMP(4)	8	74588	1311.71	12395	445.23

Table 7.1: Verifying the mutual exclusion property of Lamport's mutual exclusion algorithm.

7.11 Parallelization issues

The integer programming algorithm described in this chapter can easily be implemented on a set of parallel processing nodes. It is enough to unfold one or more steps of the recursion and distribute the **for all** loop (see Figure 7.6) between the processors.[11]

The algorithm is appropriate for both shared and distributed memory architectures. In the former case, each processor must have its own copy of the arrays

[11] For a balanced distribution of tasks, it is better to create a queue of unprocessed recursive calls.

Problem	Unfolding				Time [s]					
	$\lvert B\rvert$	$\lvert E\rvert$	$\lvert E_{cut}\rvert$	time [s]	*McM*	*MIP*	*SM*	*CLP*		
								(7.11)	*o-t-fly*	(7.24)
Q	16123	8417	1188	6	*mem*	78549	0.31	0.06	0.06	0.06
Speed	4929	2882	1219	1	23.84	35	0.08	0.03	0.02	<0.01
Dac(6)	92	53	0	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dac(9)	167	95	0	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dac(12)	260	146	0	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dac(15)	371	206	0	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dp(6)	204	96	30	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dp(8)	368	176	56	<1	<0.01	1	0.02	<0.01	<0.01	<0.01
Dp(10)	580	280	90	<1	0.02	2	0.02	<0.01	<0.01	<0.01
Dp(12)	840	408	132	<1	0.06	3	0.02	<0.01	<0.01	<0.01
Elev(1)	296	157	59	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Elev(2)	1562	827	331	<1	0.67	8	0.03	<0.01	<0.01	<0.01
Elev(3)	7398	3895	1629	1+1	98.78	311	0.13	0.03	0.01	<0.01
Elev(4)	32354	16935	7337	28+12	*mem*	8355	0.58	0.20	0.13	0.06
Hart(25)	179	102	1	<1	<0.01	<1	0.02	<0.01	<0.01	<0.01
Hart(50)	354	202	1	<1	<0.01	<1	0.02	<0.01	<0.01	0.01
Hart(75)	529	302	1	<1	<0.01	<1	0.02	<0.01	<0.01	0.01
Hart(100)	704	402	1	<1	<0.01	1	0.02	<0.01	0.01	0.02
Key(2)	1310	653	199	<1	0.25	19	0.03	<0.01	0.01	<0.01
Key(3)	13941	6968	2911	0+1	*mem*	3008	0.33	0.28	0.24	0.05
Key(4)	135914	67954	32049	0+403	*mem*	*time*	3.45	2.98	4.81	0.30
Mmgt(1)	118	58	20	<1	0.02	<1	<0.01	<0.01	<0.01	<0.01
Mmgt(2)	1280	645	260	<1	0.25	109	0.02	<0.01	<0.01	<0.01
Mmgt(3)	11575	5841	2529	0+1	*mem*	24436	0.19	0.05	0.06	0.06
Mmgt(4)	92940	46902	20957	1+222	*mem*	*mem*	22.19	29.42	29.81	64.26
Sent(25)	383	216	40	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Sent(50)	458	241	40	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Sent(75)	533	266	40	<1	<0.01	1	<0.01	<0.01	<0.01	<0.01
Sent(100)	608	291	40	<1	<0.01	1	<0.01	<0.01	<0.01	<0.01

Table 7.2: Experimental results for deadlocked net systems.

X and *Fixed*. In the latter case, each node must have its own copies of all arrays, the system of constraints, and the prefix. Such a strategy requires relatively low amount of message passing.

If the algorithm is used for finding a shortest path, then each computing node should broadcast the value of the function $\|\cdot\|$ as soon as it finds a solutions which is better then the earlier ones. This allows the other nodes take this information into account and reduce their search spaces.

Though the algorithm does admit efficient parallelization, it might be too hasty an idea to invest effort in doing it. Indeed, the experiments in Section 7.12 show that, at least for deadlock detection, *the bottleneck is generating the prefix rather than model checking it.* Thus the development of efficient unfolding algorithms is a more pressing issue (see Chapters 4–6).

Having said that, the possibility of parallelization of this algorithm still may be important, e.g., for the model checking problem described in Chapter 8.

7.12 Experimental results

For testing the performance of the proposed algorithm we used the benchmarks described in Chapter 3 and the LAMP(n) series of examples described in Chapter 6. The results of experiments are summarized in Tables 7.2–7.4.

The meaning of the columns in the Tables 7.2 and 7.3 is as follows (from left to right): the name of the problem; the number of conditions, events and cut-off events in the canonical prefix; the time spent by the unfolding algorithm described in Chapters 2, 4, and 5 to generate the canonical prefixes;[12] the time taken by the DLCHECK deadlock checker based on McMillan's method (see [74,77]), the PEP2LP tool based on the *MIP* algorithm (see [77]), and the MCSMODELS tool based on computing stable models of a logic program (see [40–42,44]); the time taken by the standard and the on-the-fly versions of the new algorithm based on (7.11), and the time taken by the new algorithm based on (7.24). We use 'time' to indicate that the test had not stopped after 15 hours,[13] 'mem' to indicate that the test terminated because of memory overflow, and 'inst' to indicate that the test gave an incorrect result or terminated because of numerical instability. DLCHECK, PEP2LP, and MCSMODELS were taken from the distribution of the PEP tool.

Clearly, the performance of the *MIP* algorithm highly depends on the performance of the tool used to solve the system of constraints.[14] In the experiments, we used the LP-SOLVE general purpose LP-solver by M.R.C.M.Berkelaar, since the CPLEX^{TM} tool used in [77] is commercial. As CPLEX^{TM} is considered to be more powerful than LP-SOLVE, the results in the *MIP* column could be better.

Table 7.4 contains the results of executing the algorithm in Section 7.4 using (7.11) and (7.24) as systems of constraints for deadlock detection. The meaning of the columns is as follows (from left to right): the name of the problem; the number of constraints and monomials in the system of constraints (7.11); the number of explored Σ^{Θ}-compatible vectors and the time spent by the algorithm while solving (7.11); the time spent by the on-the-fly version of the algorithm while solving (7.11); the number of constraints and monomials in the system of constraints (7.24); the number of explored Σ^{Θ}-compatible vectors and the time spent by the algorithm while solving (7.24). Note that for the on-the-fly method only time is given, since it does not explicitly generate the system of constraints and the number of explored Σ^{Θ}-compatible vectors is exactly the same as for the standard version of the algorithm.

One can see that the search space is usually (but not always!) greater for (7.24), because it does not allow as efficient optimizations as (7.11), but

[12]In all cases there was only one thread created, so no parallelism was used. The concurrency relation was abandoned after first 20000 generated events.

[13]Some of the experiments were run over a weekend, so the time shown in the tables can actually be greater.

[14]In fact, the proposed algorithm can be considered as a specialized solver for (7.8), since the partial order and the conflict relation can be reconstructed from the constraints $M_{in} + \mathfrak{I}_{\Sigma^{\Theta}} \cdot x \geq 0$.

since the size of the system (7.24) is often smaller, the actual running time of the algorithm is still acceptable. Moreover, memory savings for some of the examples are very significant. In view of the results in Table 7.4, the on-the-fly approach has a clear advantage, since it is not much slower than the standard method, but uses much less memory and is easier to implement.

Although the performed testing was limited in scope, it appears that the algorithm proposed in this chapter is fast, even for large prefixes. In [77], it has been pointed out that the *MIP* approach is good for 'wide' prefixes with a high number of cut-off events, whereas for prefixes with a small percentage of cut-off events, McMillan's approach is better. It turns out that the proposed approach works well for both 'wide' prefixes with a high number of cut-off events and conflicts and 'narrow' ones with a high number of causal dependencies. The worst case is a prefix with a small number of conflicts and partial order dependencies (i.e., when nearly all pairs of events are in the *co* relation), combined with a small percentage of cut-off events. As the deadlock detection problem is NP-complete in the size of prefix, such examples can be artificially constructed (see, e.g., [75], where a reduction from the *3-SAT* problem is given), but we expect that the new algorithm should work well for practical verification problems.

Among the tested algorithms, the only comparable method was that based on the translation of a deadlock detection problem into a problem of finding a stable model of a logic program, proposed in [40–42, 44]; the problem was then solved using the SMODELS tool (see [79]). After discussing this approach with its author, we concluded that if the logic solver used is powerful enough to model 'downclosing' of configurations and freezing conflicting events in linear time (and this is the case for SMODELS) then the timing results of the *SM* method and the new algorithm applied to the system (7.11) should be of the same order of magnitude. Indeed, the experimental results confirm that both methods are comparable, though they are based on different principles.

7.13 Conclusions

Experimental results indicate that the algorithm we proposed in this chapter can easily solve problems with hundreds of thousands of variables (see, e.g., the LAMP(4) example). This overcomes the existing limitations, as *MIP* problems with even a few hundreds of integer variables are often a hard task for general purpose solvers. It is worth emphasizing that earlier the limitation was not the size of computer memory, but rather the time to solve an NP-complete problem. With the proposed method, the main limitation was rather the size of memory to store the system of constraints, but the on-the-fly approach overcomes this problem. Therefore, building prefixes becomes the bottleneck for the verification of deadlock freeness of Petri nets (see Tables 7.2 and 7.3), and in future research we will aim at developing more efficient algorithms for constructing large unfoldings.

Problem	Unfolding				Time [s]					
	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	time [s]	*McM*	*MIP*	*SM*		*CLP*	
								(7.11)	*o-t-fly*	(7.24)
Abp	337	167	56	<1	0.06	2	<0.01	<0.01	<0.01	<0.01
Bds	12310	6330	3701	1	*mem*	4240	0.25	0.13	0.13	0.06
Buf(100)	10101	5051	1	6	0.02	*mem*	0.05	0.01	0.01	0.02
Byz	42276	14724	752	85	*mem*	6802	8.08	4.58	5.33	1.14
Ftp	178085	89046	35197	1336	*mem*	*mem*	4.28	5.22	7.28	10.75
Cyclic(3)	52	23	4	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Cyclic(6)	112	50	7	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Cyclic(9)	172	77	10	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Cyclic(12)	232	104	13	<1	0.02	1	<0.01	<0.01	<0.01	<0.01
Dme(2)	487	122	4	<1	0.03	5	<0.01	<0.01	<0.01	<0.01
Dme(3)	1210	321	9	<1	0.52	197	0.03	<0.01	<0.01	0.01
Dme(4)	2381	652	16	<1	3.98	40	0.09	0.02	0.01	0.01
Dme(5)	4096	1145	25	1	30.58	114	0.20	0.05	0.05	0.05
Dme(6)	6451	1830	36	1	203	*inst*	0.58	0.11	0.13	0.13
Dme(7)	9542	2737	49	3	1000	*inst*	1.50	0.30	0.33	0.34
Dme(8)	13465	3896	64	8	3960	*inst*	4.06	0.67	0.80	0.91
Dme(9)	18316	5337	81	17	13090	*inst*	10.64	1.56	1.89	2.34
Dme(10)	24191	7090	100	33	37770	*inst*	27.59	3.66	4.44	6.09
Dme(11)	31186	9185	121	61	97550	*inst*	69.11	8.42	10.27	15.30
Dpd(4)	594	296	81	<1	0.39	7	0.02	<0.01	0.01	<0.01
Dpd(5)	1582	790	211	<1	21.97	59	0.05	0.02	0.01	0.01
Dpd(6)	3786	1892	499	<1	546	823	0.17	0.06	0.03	0.03
Dpd(7)	8630	4314	1129	1	11702	12032	0.52	0.27	0.17	0.14
Dpfm(2)	12	5	2	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dpfm(5)	67	31	20	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Dpfm(8)	426	209	162	<1	0.05	1	<0.01	<0.01	<0.01	0.01
Dpfm(11)	2433	1211	1012	1	5.33	727	0.03	<0.01	0.01	<0.01
Dph(4)	680	336	117	<1	0.42	5	0.02	<0.01	<0.01	<0.01
Dph(5)	2712	1351	547	<1	66.48	221	0.06	0.03	0.02	0.02
Dph(6)	14590	7289	3407	1	*mem*	28941	0.78	0.44	0.41	0.47
Dph(7)	74558	37272	19207	40	*mem*	*time*	8.53	5.33	7.08	13.33
Furn(1)	535	326	189	<1	0.11	1	<0.01	<0.01	<0.01	<0.01
Furn(2)	4573	2767	1750	1	174	410	0.06	0.03	0.03	0.06
Furn(3)	30820	18563	12207	4	*mem*	139371	0.66	0.36	0.39	1.58
Gasnq(2)	338	169	46	<1	0.08	3	<0.01	<0.01	<0.01	<0.01
Gasnq(3)	2409	1205	401	<1	62.70	1119	0.08	0.03	0.05	0.05
Gasnq(4)	15928	7965	2876	3	*mem*	*time*	1.42	0.94	1.36	1.39
Gasnq(5)	100527	50265	18751	389	*mem*	*mem*	34.83	39.76	64.40	80.12
Gasq(1)	43	21	4	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Gasq(2)	346	173	54	<1	0.09	3	0.02	<0.01	<0.01	<0.01
Gasq(3)	2593	1297	490	<1	94.59	1085	0.08	0.03	0.05	0.03
Gasq(4)	19864	9933	4060	6	*mem*	*time*	1.48	1.19	1.72	1.59
Lamp(2)	711	368	102	<1	0.75	19	0.02	<0.01	<0.01	<0.01
Lamp(3)	23424	12026	4562	6	*mem*	*time*	2.19	0.77	1.08	1.25
Lamp(4)	736507	375983	167780	28243	*time*	*mem*	322	10122	811	1257
Over(2)	83	41	10	<1	<0.01	0	<0.01	<0.01	<0.01	<0.01
Over(3)	369	187	53	<1	0.05	3	<0.01	<0.01	<0.01	<0.01
Over(4)	1536	783	237	<1	11.72	151	0.02	<0.01	0.01	0.02
Over(5)	7266	3697	1232	1	1386	16934	0.14	0.06	0.05	0.22
Ring(3)	97	47	11	<1	<0.01	<1	<0.01	<0.01	<0.01	<0.01
Ring(5)	339	167	37	<1	0.05	5	<0.01	<0.01	<0.01	<0.01
Ring(7)	813	403	79	<1	0.83	120	0.02	<0.01	0.01	0.01
Ring(9)	1599	795	137	<1	13.17	4696	0.08	0.03	0.01	0.06
Rw(6)	806	397	327	<1	0.12	1	0.02	<0.01	<0.01	<0.01
Rw(9)	9272	4627	4106	<1	172	37142	0.11	0.01	0.05	0.03
Rw(12)	98378	49177	45069	5	*mem*	*mem*	1.28	0.80	3.66	0.48
Sync(2)	3884	2091	474	<1	237	2846	0.08	0.05	0.06	0.11
Sync(3)	28138	15401	5210	8	*mem*	*time*	1.36	1.47	1.69	3.53

Table 7.3: Experimental results for deadlock-free net systems.

Problem	CLP – (7.11)					CLP – (7.24)			
	System size		Performance			System size		Performance	
	cons	mono	vec exp	$t[s]$	o-t-fly t[s]	cons	mono	vec exp	$t[s]$
Bds	6330	89036	32	0.13	0.13	59	10633	91	0.06
Buf(100)	5051	14951	1	0.01	0.01	101	14951	1	0.02
Byz	14724	1609871	2240	4.58	5.33	274	48761	176	1.14
Ftp	89046	5325083	123	5.22	7.28	228	326279	291	10.75
Q	8417	153828	8	0.06	0.06	134	24776	16	0.06
Speed	2882	56712	12	0.03	0.02	39	8006	17	<0.01
Dme(2)	122	528	3	<0.01	<0.01	78	434	5	<0.01
Dme(3)	321	1890	11	<0.01	<0.01	117	1119	18	0.01
Dme(4)	652	5536	31	0.02	0.01	156	2228	52	0.01
Dme(5)	1145	14250	74	0.05	0.05	195	3845	131	0.05
Dme(6)	1830	32832	160	0.11	0.13	234	6054	305	0.13
Dme(7)	2737	68698	334	0.30	0.33	273	8939	687	0.34
Dme(8)	3896	132480	684	0.67	0.80	312	12584	1517	0.91
Dme(9)	5337	238626	1386	1.56	1.89	351	17073	3307	2.34
Dme(10)	7090	406000	2792	3.66	4.44	390	22490	7145	6.09
Dme(11)	9185	658482	5606	8.42	10.27	429	28919	15335	15.30
Dpd(4)	296	1630	23	<0.01	0.01	36	800	26	<0.01
Dpd(5)	790	6028	38	0.02	0.01	45	2142	43	0.01
Dpd(6)	1892	21970	62	0.06	0.03	54	5148	71	0.03
Dpd(7)	4314	84318	114	0.27	0.17	63	11758	130	0.14
Dph(4)	336	1739	21	<0.01	<0.01	41	849	37	<0.01
Dph(5)	1351	10456	48	0.03	0.02	61	3423	107	0.02
Dph(6)	7289	104719	134	0.44	0.41	85	18851	405	0.47
Dph(7)	37272	1275219	377	5.33	7.08	113	98002	1533	13.33
Elev(1)	157	770	3	<0.01	<0.01	67	572	3	<0.01
Elev(2)	827	7725	5	<0.01	<0.01	191	3729	5	<0.01
Elev(3)	3895	78040	5	0.03	0.01	484	21958	5	<0.01
Elev(4)	16935	815397	6	0.20	0.13	1173	123035	6	0.06
Furn(1)	326	1453	8	<0.01	<0.01	37	766	17	<0.01
Furn(2)	2767	16369	38	0.03	0.03	63	7618	91	0.06
Furn(3)	18563	168732	147	0.36	0.39	93	59258	294	1.58
Gasnq(2)	169	689	19	<0.01	<0.01	77	565	23	<0.01
Gasnq(3)	1205	11594	205	0.03	0.05	205	5076	264	0.05
Gasnq(4)	7965	256089	1792	0.94	1.36	433	41425	2496	1.39
Gasnq(5)	50265	7014616	15684	39.76	64.40	791	316717	23537	80.12
Gasq(1)	21	53	1	<0.01	<0.01	19	53	2	<0.01
Gasq(2)	173	729	18	<0.01	<0.01	85	608	20	<0.01
Gasq(3)	1297	14371	197	0.03	0.05	409	8408	215	0.03
Gasq(4)	9933	480899	1541	1.19	1.72	2313	181926	1811	1.59
Key(2)	653	3487	18	<0.01	0.01	82	1335	16	<0.01
Key(3)	6968	109416	35	0.28	0.24	119	12612	31	0.05
Key(4)	67954	4918109	9	2.98	4.81	156	119926	15	0.30
Lamp(2)	368	1795	34	<0.01	<0.01	56	907	42	<0.01
Lamp(3)	12026	164024	779	0.77	1.08	104	27314	794	1.25
Lamp(4)	375983	24761671	15421	10122	811	166	803074	15325	1257
Mmgt(1)	58	172	2	<0.01	<0.01	58	172	2	<0.01
Mmgt(2)	645	5591	7	<0.01	<0.01	114	2375	11	<0.01
Mmgt(3)	5841	224779	88	0.05	0.06	172	25350	138	0.06
Mmgt(4)	46902	9178373	6360	29.42	29.81	232	234258	10818	64.26
Rw(6)	397	2965	1	<0.01	<0.01	85	1255	12	<0.01
Rw(9)	4627	141567	1	0.01	0.05	181	14059	18	0.03
Rw(12)	49177	8501695	1	0.80	3.66	313	147877	24	0.48
Sync(2)	2091	15321	117	0.05	0.06	88	6301	248	0.11
Sync(3)	15401	488170	684	1.47	1.69	141	48239	1060	3.53

Table 7.4: Comparison of the deadlock detection methods based on (7.11) and (7.24).

Chapter 8

Detecting State Coding Conflicts in STGs

In this chapter, we apply the technique described in Chapter 7 to a specific problem arising in synthesis of asynchronous circuits, namely detecting state coding conflicts in their specifications.

Signal Transition Graphs (STGs) is a formalism widely used for describing the behaviour of asynchronous control circuits. Typically, they are used as a specification language for the synthesis of such circuits (see, e.g., [11,92]). STGs are a form of interpreted Petri nets, in which transitions are labelled with the names of rising and falling edges of circuit signals. Circuit synthesis based on STGs involves: (i) checking the necessary and sufficient conditions for an STG's implementability as a logic circuit; (ii) modifying, if necessary, the initial STG to make it implementable; and (iii) finding appropriate boolean covers for the next-state functions of output and internal signals and obtaining them in the form of boolean equations for the logic gates of the circuit. One of the commonly used STG-based synthesis tools, PETRIFY (see [21]), performs all of these steps automatically, after first constructing the reachability graph of the initial STG specification. A vivid example of its use is the design of many circuits for the AMULET-3 microprocessor (see, e.g., [36]). Since popularity of this tool is steadily growing, it is very likely that STGs and Petri nets will increasingly be seen as an intermediate (back-end) notation for the design of large controllers.

In order to increase efficiency, PETRIFY uses symbolic (BDD-based) techniques to represent the reachable state space and to capture important relationships (e.g., excitation and quiescent regions, concurrency and conflict relations). While this purely state-based approach is very convenient for finding good synthesis solutions, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits, which are often restrictive, especially if the STG models are not constructed by a human designer but rather generated automatically from high-level hardware descriptions.

In order to alleviate this problem, Petri net analysis techniques based on

causal partial order semantics, in the form of Petri net unfoldings, were applied for circuit synthesis in [86, 87]. The idea behind the approach described there was to work with approximate boolean covers obtained for structural elements of the unfolding, namely conditions and events, as opposed to the use of exact boolean covers for markings and excitation regions extracted from the reachability graph. Although the results were still preliminary, they demonstrated, for some examples, a clear superiority — in terms of memory and time efficiency — of the unfolding-based approach. The main shortcoming of the work described in [86,87] was that its approximation and refinement strategy was fairly straightforward and could not cope well with the 'don't care' state subsets, i.e., sets of states which would have been unreachable if the exact reachability analysis was applied (see [65]). Bearing this in mind, there is a clear need for further advancement of the unfolding-based methods, both in theory and algorithms, for solving the above mentioned synthesis tasks. In this chapter, we propose a solution for one of the subproblems, central to the implementability analysis in step (i), viz. checking the Complete State Coding (CSC) and the Unique State Coding (USC) conditions (see, e.g., [11]). In essence, this problem consists in detecting the state coding conflicts, which occur when semantically different reachable states have the same binary encoding.

The CSC (and USC) problem is often seen as one which consists of two parts: the detection of coding conflicts, and the elimination of such conflicts. The second part may be addressed, for example, by means of changing the causality or ordering constraints (i.e., adding extra places and arcs in the STGs to make implicit timing assumptions explicit), or by introducing 'additional memory' into the system in the form of internal signals, which can be done on the level of complete prefix, without building the reachability graph (see [73]). The latter approach typically requires behaviour-preserving (with respect to the original set of events) transformations, which are more difficult than simply adding new ordering constraints. A number of methods for solving the CSC problem are available (see, e.g., [22], for a brief review). Most of them work in the state graph framework and are general in terms of applicability to the widest possible class of STGs (with bounded underlying Petri Nets). Some, such as [91], operate directly on the STG level, but they restrict the class of the underlying Petri nets to, e.g., marked graphs.

The application of Petri Net unfoldings to the detection of state conflicts in an STG was first attempted in [65]. That work has advanced the ideas of slices and cover approximations of [86, 87], and presented theory and algorithms for 'fast' and 'refined' detection of coding conflicts. However, those algorithms have not yet been implemented and proved efficient in experiments, and in their 'refinement' part they still require the construction of the (partial) state space for the subsets of unfolding cuts that evaluate a given boolean cover to true.

In this chapter, we investigate another kind of unfolding-based approach. We show that the notion of a state conflict can be characterized in terms of a system of integer constraints, and the technique developed in Chapter 7 can

then be used to efficiently solve it. In addition to the CSC and USC problems, the integer programming approach can easily be modified to check the normalcy property of STGs, which is a necessary condition for their implementability using logic gates whose characteristic functions are monotonic.

The proposed new approach to state coding conflict detection is in some sense opposite to the state graph based one, and exploits only the characteristics of the unfolding structure itself. Unlike [65], this method is not concerned with boolean covers for parts of the unfolding.

The initial motivation for applying this technique to the problem of CSC (and USC) comes from its success in speeding up deadlock detection, which has subsequently been extended to solving other model checking problems (see Chapter 7). The results of initial experiments demonstrate that the proposed algorithm is not only memory-efficient, but also can in many cases achieve significant speedups. It is also worth pointing out that the method allows one not only to find conflicting reachable states, but also to derive execution paths leading to them without performing a reachability analysis.

This chapter is based on the results developed in [62, 63].

8.1 Basic definitions

In this section, we present basic definitions concerning STGs and their unfoldings (see also [21, 22, 64, 65, 86, 87, 89, 92]).

8.1.1 Signal Transition Graphs

A *Signal Transition Graph (STG)* is a quadruple $\Gamma \stackrel{\text{df}}{=} (\Sigma, Z, \lambda, v^0)$ such that $\Sigma = (N, M_0)$ is a net system, $Z \stackrel{\text{df}}{=} \{z_1, \ldots, z_k\}$ is a finite set of *signals* which generate a finite alphabet $Z^{\pm} \stackrel{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, $\lambda : T \to Z^{\pm} \cup \{\tau\}$ is a *labelling function*, and $v^0 \in \{0, 1\}^k$ is a *vector of initial signal values*. Here $\tau \notin Z^{\pm}$ is a label indicating a 'dummy' transition, which does not change the value of any signal. The signal transition labels are of the form z^+ or z^-, and denote the transitions of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. Signal transitions are associated with the actions which change the value of a particular signal. We will also use the notation $z^{\pm}$ to denote a transition of signal z if we are not particularly interested in its direction. The labelling function λ can be generalized to (finite or infinite) sequences of transitions as follows: $\lambda(t_1 t_2 \ldots t_i[\ldots]) = \lambda(t_1)\lambda(t_2)\ldots\lambda(t_i)[\ldots]$. Γ inherits the operational semantics of its underlying net system Σ, including the notions of transition enabling and execution, and firing sequences.

With a finite sequence of transitions σ we associate an integer *signal change vector* $v^{\sigma} \stackrel{\text{df}}{=} (v_1^{\sigma}, v_2^{\sigma}, \ldots, v_k^{\sigma}) \in \mathbb{Z}^k$, so that each v_i^{σ} is the difference between the number of the occurrences of z_i^+-labelled and z_i^--labelled transitions in σ. A *state* of Γ is a pair (M, v), where M is a marking of Σ and $v \in \mathbb{Z}^k$. We

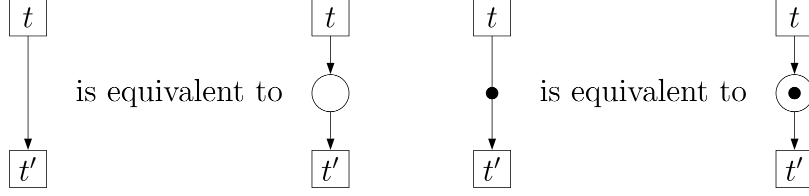

Figure 8.1: The interpretation of transition-transition arcs in figures.

denote by $\mathcal{S}(\Gamma) \stackrel{\text{df}}{=} \mathcal{M}(N) \times \mathbb{Z}^k$ the set of possible states of Γ, where $\mathcal{M}(N)$ is the set of possible markings of the net underlying Γ. The transition relation on $\mathcal{S}(\Gamma) \times T \times \mathcal{S}(\Gamma)$ is defined as $(M', v') \xrightarrow{t} (M'', v'')$ iff $M'[t\rangle M''$ and $v'' = v' + v^t$.

For a finite sequence of transitions $\sigma = t_1 \dots t_i$, we write $s' \xrightarrow{\sigma} s''$ if there are states $s_1, \dots, s_{i+1}$ such that $s_1 = s'$, $s_{i+1} = s''$, and $s_j \xrightarrow{t_j} s_{j+1}$, for all $j \in \{1, \dots, i\}$. If the identity of s'' is irrelevant, we write $s' \xrightarrow{\sigma}$ to denote that $s' \xrightarrow{\sigma} s''$ for some $s'' \in \mathcal{S}(\Gamma)$. In these definitions, we allow σ to be not only a sequence of transitions, but also a sequence of elements of $Z^{\pm} \cup \{\tau\}$; in such a case, $s' \xrightarrow{\sigma} s''$ means that $s' \xrightarrow{\sigma'} s''$ for some sequence of transitions σ' such that $\lambda(\sigma') = \sigma$.

The *state graph* of Γ is a triple $SG_\Gamma \stackrel{\text{df}}{=} (S, A, s_0)$ such that the set of *reachable states* S is the smallest (w.r.t. $\subseteq$) closed under the transition relation set containing the *initial state* $s_0 \stackrel{\text{df}}{=} (M_0, v^0)$, and the set of *arcs* A is the restriction of the transition relation to $S \times T \times S$. The *state assignment function* $Code : S \to \mathbb{Z}^k$ is defined as $Code((M, v)) \stackrel{\text{df}}{=} v$.

Γ is *consistent* if, for every reachable state $s \in S$, $Code(s) \in \{0, 1\}^k$, i.e., for every finite execution sequence σ of Σ starting at the initial state, $v^0 + v^\sigma \in \{0, 1\}^k$. Such a property guarantees that, for every signal $z \in Z$, the STG satisfies the following two conditions: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has always the same sign (either rising or falling); and (ii) the rising and falling labels z alternate in any firing sequence of Γ. In this chapter it is assumed that all the considered STGs are consistent. (The consistency of an STG can easily be checked during the process of building its finite and complete prefix, e.g., as described in [86].)

In addition to the drawing conventions described in Section 1.3, we use the following one. When an arc in a figure connects two transitions, it is assumed that there is a place 'in the middle' of the arc. Moreover, an arc with a token on it is interpreted similarly, but the place contains a token. Figure 8.1 illustrates these conventions.

Two distinct states s' and s'' of SG_Γ are in *USC conflict* if $Code(s') = Code(s'')$. Γ satisfies the *Unique State Coding (USC)* property if no two states of SG_Γ are in USC conflict.

It is often the case that an STG's signals are partitioned into input signals, Z_I, and output signals, Z_O (the latter may also include internal signals). Input

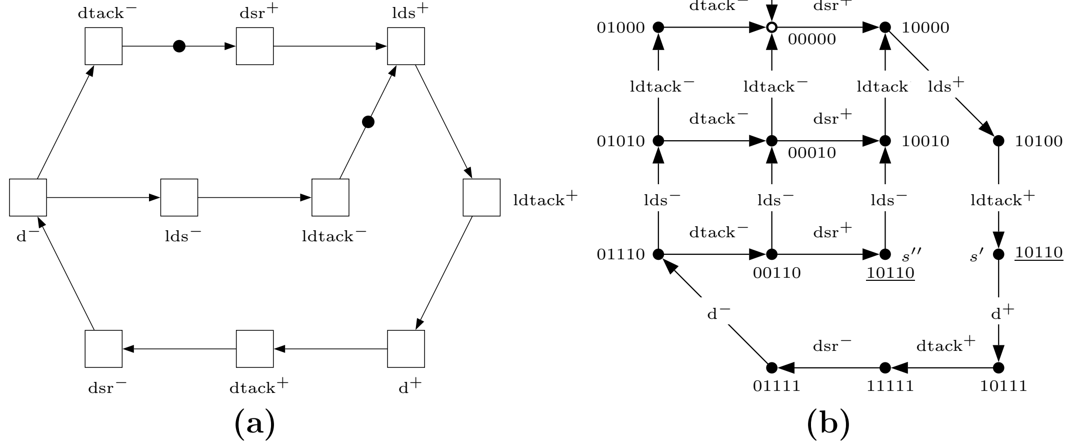

Figure 8.2: An STG modelling a simplified VME bus controller **(a)** and its state graph with a CSC conflict between two states **(b)**. The order of signals in the binary codes is: *dsr*, *dtack*, *lds*, *ldtack*, *d*.

signals are assumed to be generated by the environment, while output signals are produced by the logical gates in the circuit. The problem of logic synthesis consists in deriving boolean equations for the output signals, which requires the conditions for enabling output signal transitions in the state graph of the STG to be defined without ambiguity by the encoding of each reachable state.

To capture this, we define the set of enabled output signals at a state s as $Out(s) \stackrel{\text{df}}{=} \{z \in Z_O \mid s \xrightarrow{\tau^* z^{\pm}}\}$. Two distinct states s' and s'' of SG_Γ are in *CSC conflict* if $Code(s') = Code(s'')$ and $Out(s') \neq Out(s'')$. Γ satisfies the *Complete State Coding (CSC)* property if no two states of SG_Γ are in CSC conflict. Clearly, if Γ has USC then it also has CSC.

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark) is shown in Figure 8.2(a). Part (b) of this figure illustrates CSC conflict between two different states, s' and s'', that have the same code, 10110, but $Out(s') = \{d\} \neq Out(s'') = \{lds\}$.

8.1.2 STG branching processes

A *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda, v_0)$ is a branching process of Σ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm} \cup \{\tau\}$. With any finite set of events $E' \subseteq E$, we associate an integer *signal change vector* $v^{E'} \stackrel{\text{df}}{=} (v_1^{E'}, \ldots, v_k^{E'}) \in \mathbb{Z}^k$, such that each $v_i^{E'}$ is the difference between the number of z_i^+-labelled and z_i^--labelled events in E'. One can easily prove that $v^{E'} = v^{\sigma}$, where σ is an arbitrary linearization of $h\{\!|E'|\!\}$. The function *Code* is then extended to finite configurations of the branching process of Γ through $Code(C) \stackrel{\text{df}}{=} v^0 + v^C$. Note that $Code(C) = v^0 + v^{\sigma}$, for any linearization σ of $h\{\!|C|\!\}$, i.e., this definition is consistent with the definition of *Code* for SG_Γ

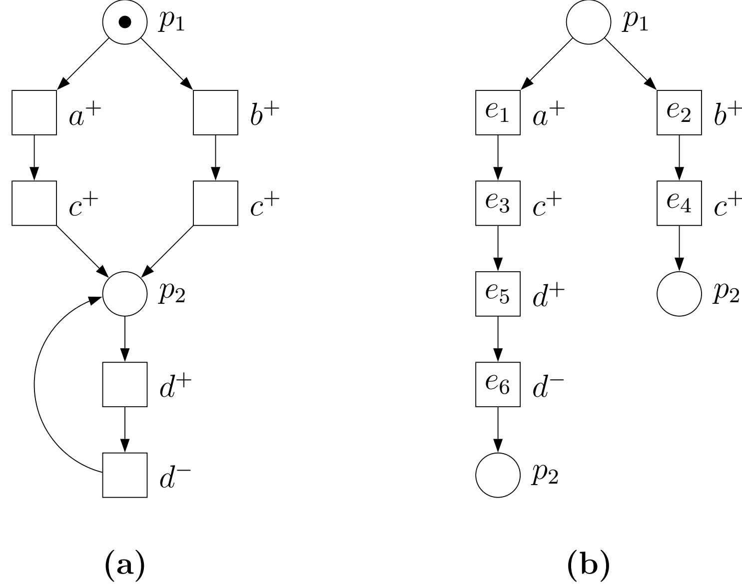

Figure 8.3: A consistent STG **(a)** and its canonical (w.r.t. Θ_{ERV}) prefix **(b)**. Note that e_4 is a cut-off event and a reachable state with the code $\{a \mapsto 0, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$ is not represented in it.

in the sense that the pair $(Mark(C), Code(C))$ is a state of SG_Γ, and for any state s of SG_Γ there exists a configuration C in the unfolding of Γ such that $s = (Mark(C), Code(C))$.

It is important to note that, since the states of an STG do not necessarily correspond to its reachable markings, some of the states may be not represented in the prefixes built using the cutting context Θ_{ERV} (see Figure 8.3). It was suggested in [86] to restrict the cut-off criterion by requiring that not only final markings of the two configurations should be equal, but also their codes. This idea can be formalized by choosing the equivalence relation $\approx_{code}$ rather than $\approx_{mar}$ in the cutting context (see Chapter 2).

8.2 State coding conflict detection using integer programming

In the rest of this chapter, we assume that Θ is a dense cutting context with the equivalence relation $\approx_{code}$, $\Gamma = (\Sigma, Z, \lambda)$ is a consistent and bounded STG, and $\Gamma^\Theta \overset{\mathrm{df}}{=} (B, E, G, M_{in})$ is the safe net system built from the canonical prefix $Unf_\Gamma^\Theta = (B, E, G, h)$ of the unfolding of Γ, where M_{in} is the canonical initial marking of Γ^Θ which places a single token in each of the minimal conditions and

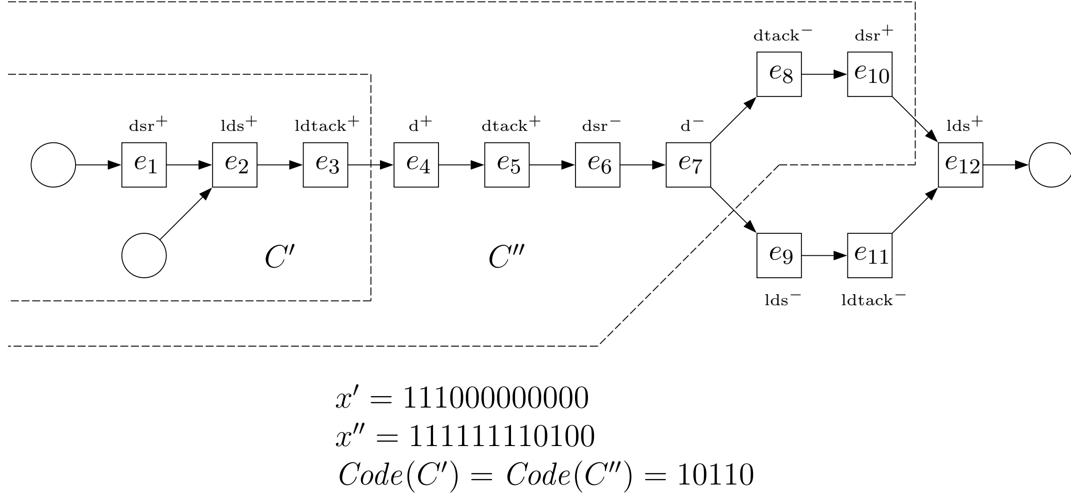

Figure 8.4: Unfolding prefix for the VME bus example and CSC conflict between configurations C' and C'' (encoded by the vectors x' and x'') corresponding to the states s' and s'' in Figure 8.2(b). The order of signals in the binary codes is: dsr, $dtack$, lds, $ldtack$, d.

no token elsewhere.[1] The set of cut-off events of Unf_{Γ}^{Θ} will be denoted by E_{cut}. Until Section 8.5, we assume that Γ contains no τ-labelled transitions.

Suppose that two distinct reachable states (M', v') and (M'', v'') of Γ are in CSC (or USC) conflict. Then these markings are represented in Unf_{Γ}^{Θ} as some configurations C' and C'' without cut-off events such that $(M', v') = (Mark(C'), Code(C'))$ and $(M'', v'') = (Mark(C''), Code(C''))$. Let x' and x'' be respectively the Parikh vectors of C' and C''. Using the theory developed in Chapter 7, we will now transform the problem of checking the CSC (or USC) property into an integer programming problem expressed in terms of x' and x''. Note that since these variables denote Parikh vectors of configurations, they are from the domain $\{0, 1\}^{|E|}$. The constraints constituting the system to be solved are described below.

8.2.1 Encoding constraint

As described in the previous section, for a configuration C, its signal encoding vector $Code(C)$ is defined as

$$Code(C) \stackrel{\text{df}}{=} v^0 + v^C \ .$$

The above expression is a linear function of the Parikh vector x^C of C; we will denote it by $Code(x^C)$. With this notation, the condition that the final encodings of two configurations given by their Parikh vectors x' and x'' are the same can

[1] We will often identify Γ^{Θ} and Unf_{Γ}^{Θ}, provided that this does not create an ambiguity.

be expressed as the linear constraint

$$Code(x') = Code(x'') \ . \tag{8.1}$$

Note that the value of v^0 is not needed to build it.

Figure 8.4 illustrates a CSC conflict in the unfolding prefix of the STG shown in Figure 8.2. Constraint (8.1) has the form:

$$\begin{cases} x'_1 - x'_6 + x'_{10} & = & x''_1 - x''_6 + x''_{10} & (dsr) \\ x'_5 - x'_8 & = & x''_5 - x''_8 & (dtack) \\ x'_2 - x'_9 + x'_{12} & = & x''_2 - x''_9 + x''_{12} & (lds) \\ x'_3 - x'_{11} & = & x''_3 - x''_{11} & (ldtack) \\ x'_4 - x'_7 & = & x''_4 - x''_7 & (d) \end{cases}$$

8.2.2 USC separating constraint

We are not interested in solutions with $M' = M''$, and so the *separating constraint* $M' \neq M''$ has to be added to the system. Section 7.6 provides a way of rendering it as a constraint $Mark(x') \neq Mark(x'')$ specified on Γ^{Θ}-compatible vectors. At this point, the straightforward application of the theory developed in Chapter 7 yields a full formulation of the USC conflict detection problem using the integer programming framework, together with an algorithm for solving it. The system of constraints has the form

$$\begin{cases} Code(x') = Code(x'') \\ Mark(x') \neq Mark(x'') \\ x'(e) = x''(e) = 0 \text{ for all } e \in E_{cut} \\ x', x'' \in \{0,1\}^{|E|} \text{ are } \Gamma^{\Theta}\text{-compatible} \ . \end{cases} \tag{8.2}$$

Note that since the second constraint in this system is not linear, the branching condition (7.18) cannot be applied. To make it linear, we can replace it by $Mark(x') <_{lex} Mark(x'')$, where $<_{lex}$ is the lexicographical order on integer vectors. Since Γ is a bounded STG, this constraint is linear in x' and x''. Indeed, if every place of the STG can hold at most i tokens then this constraint is equivalent to

$$\sum_{j=1}^{|P|} i^{j-1} Mark_{p_j}(x') < \sum_{j=1}^{|P|} i^{j-1} Mark_{p_j}(x'')$$

and is very similar to the comparison of two i-ary numbers; in particular, when Γ is safe, $Mark(x')$ and $Mark(x'')$ can be seen as two binary numbers.

8.2.3 CSC separating constraint

The separating constraint is more complicated if we verify the CSC rather than USC property. In order to exclude the solutions for which the sets of the enabled outputs are the same, one should add to the system (8.2) the separating constraint $Out(x') \neq Out(x'')$, where $Out(x)$ is a function computing the

enabled outputs of the state represented by the configuration given by a Γ^{Θ}-compatible vector x. (Note that we do not necessarily have to remove the constraint $Mark(x') \neq Mark(x'')$, since it holds whenever $Out(x') \neq Out(x'')$ does.) To check whether $Out(x') \neq Out(x'')$ holds for particular Parikh vectors x' and x'' one can compute $M' = Mark(x')$ and $M'' = Mark(x'')$. If $M' = M''$ then $Out(x') \neq Out(x'')$ cannot hold, and thus no further computation is needed. Otherwise, the outputs enabled by markings M' and M'' can be found directly from the STG.

Since the constraint $Out(x') \neq Out(x'')$ is, in general, not linear, one has to deal with a non-linear integer programming problem. Though the branching condition (7.18) cannot be applied in this case, the algorithm described in Chapter 7 often terminates in reasonable time.

8.2.4 Retrieving a solution

What one often wants to see as a solution is two execution sequences of the original STG, leading to states which are in CSC (or USC) conflict. BDD-based methods (see, e.g., [64, 84]) compute the characteristic function of the set of all solutions, but do not provide the information about execution sequences. Therefore, in order to retrieve them, one has to either perform a separate reachability analysis or do some kind of bookkeeping while building a BDD. With the new algorithm this information is easily available (see Chapter 7).

8.3 Verifying the normalcy property

The property of *normalcy* is a necessary condition for an STG to be implementable as a logic circuit built of gates whose characteristic functions are monotonic. The latter in turn guarantees that the circuit is *speed-independent* (i.e., it correctly operates irrespectively of delays in the gates) without the necessity to neglect (quite unrealistically) the delays in input inverters (see [89]).

Let $\Gamma = (\Sigma, Z, \lambda)$ be an STG. Normalcy is specified with respect to an output signal $z \in Z_O$, and can be given in terms of boolean *next-state function* Nxt_z defined for the reachable states. If $s = (M, v)$ is a reachable state of Γ then: $Nxt_z(s) \stackrel{\text{df}}{=} 0$ if $v_z = 0$ and no z^+-labelled transition is enabled at s, or $v_z = 1$ and a z^--labelled transition is enabled at s; and $Nxt_z(s) \stackrel{\text{df}}{=} 1$ if $v_z = 1$ and no z^--labelled transition is enabled at s, or $v_z = 0$ a z^+-labelled transition is enabled at s.

Γ satisfies the *positive normalcy* (or *p-normalcy*) condition w.r.t. an output signal z if for every pair of reachable states s' and s'', $Code(s') \geq Code(s'')$ implies $Nxt_z(s') \geq Nxt_z(s'')$. Similarly, Γ satisfies the *negative normalcy* (or *n-normalcy*) condition w.r.t. an output signal z if for every pair of reachable states s' and s'', $Code(s') \geq Code(s'')$ implies $Nxt_z(s') \leq Nxt_z(s'')$. Finally, Γ is *normal* if each output signal is either p-normal or n-normal. It turns out that normalcy

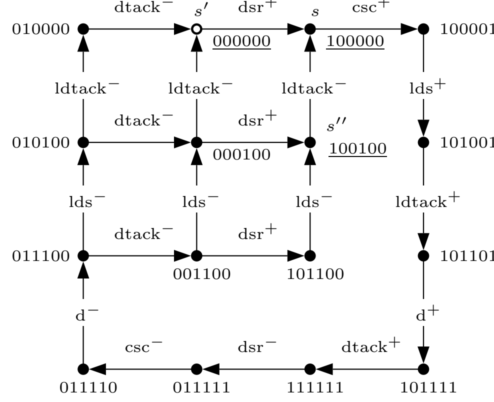

Figure 8.5: State graph which is free from CSC conflicts but shows normalcy violations (for signal *csc*) between states s, s', and s''. The order of signals in the binary codes is: *dsr*, *dtack*, *lds*, *ldtack*, *d*, *csc*.

implies CSC (see [89]).

An example of normalcy violation is illustrated in Figure 8.5. This is a state graph obtained after the CSC conflict has been resolved in the STG of Figure 8.2(a) by means of a new state signal, *csc*. The resulting model, free from CSC conflicts, is implementable — the next-state functions for all output signals are as follows: $lds = d \vee csc$, $dtack = d$, $d = ldtack \wedge csc$, and $csc = dsr \wedge (csc \vee \overline{ldtack})$. Nonetheless, normalcy is violated for signal *csc*. Indeed, $Code(s) > Code(s')$, $Nxt_{csc}(s) > Nxt_{csc}(s')$ but $Code(s) < Code(s'')$, $Nxt_{csc}(s) > Nxt_{csc}(s'')$, so *csc* is neither n-normal nor p-normal. This is reflected in the implementation function for *csc*, which is non-monotonic: it is positive w.r.t. *dsr* and negative w.r.t. *ldtack* (note that the corresponding gate has an input inverter).

The verification method described earlier in this chapter can be adopted to check the normalcy of STGs. Indeed, it is enough to solve in Γ^{Θ}-compatible vectors x', x'' the following (non-linear) system of constraints:

$$\begin{cases} Code(x') \geq Code(x'') \\ Nxt_z(x')\ R_z\ Nxt_z(x'') \text{ for each } z \in Z_O \\ x'(e) = x''(e) = 0 \text{ for all } e \in E_{cut} \\ x', x'' \in \{0,1\}^{|E|} \text{ are } \Gamma^{\Theta}\text{-compatible .} \end{cases} \tag{8.3}$$

where $R_z \in \{<, >\}$ depending on the type of normalcy of the signal z. If it is not known in advance, the algorithm can leave R_z undefined until the moment when the first constraint is satisfied and $Nxt_z(x') \neq Nxt_z(x'')$. Then it fixes R_z so that the constraint does not hold and continues the search. Alternatively, R_z's can be determined by looking at the *triggers* of an event e labelled by $z^{\pm}$, defined

as $Trig(e) = \max_{\prec}\langle e\rangle$.[2] With the exception of certain degraded cases, for each signal z there is an event e labelled by $z^{\pm}$ such that $Trig(e) \neq \emptyset$, and one can easily show the following:

- If the labels of the events in $Trig(e)$ do not all have the same sign then z is neither p-normal nor n-normal, and no further computation is required.
- If the labels of the events in $Trig(e)$ have the same sign and this sign coincides with that of $\lambda(h(e))$ then z cannot be n-normal, and thus R_z should be $>$.
- If the labels of the events in $Trig(e)$ have the same sign and this sign differs from that of $\lambda(h(e))$ then z cannot be p-normal, and thus R_z should be $<$.

8.4 The case of conflict-free nets

In many cases the performance of the proposed algorithm can be improved by exploiting specific properties of the Petri net underlying an STG Γ. For instance, if Γ is free from dynamic conflicts (in particular, this is the case for marked graphs) then the union of any two configurations of its unfolding is also a configuration. This observation can be used to reduce the search space. Indeed, according to Proposition 8.1 below, it is then enough to look only for those cases when the configurations C' and C'' being tested are ordered in the set-theoretical sense.

Proposition 8.1. *Let Γ be free from dynamic conflicts, and let C' and C'' be two finite configurations of Γ^{Θ} such that $C' \not\subseteq C''$ and $C'' \not\subseteq C'$. If the states s' and s'' represented respectively by C' and C'' are in USC / CSC / p-normalcy / n-normalcy conflict, then the state s represented by the configuration $C \stackrel{\mathrm{df}}{=} C' \cap C''$ is in respectively USC / CSC / p-normalcy / n-normalcy conflict with either s' or s''.*

Proof. We first show that

$$Code(C') \geq Code(C'') \implies Code(C') \geq Code(C) \geq Code(C'') . \qquad (*)$$

Since Γ is conflict-free, C', C'' and C are all included in the configuration $C' \cup C''$, and so each event in $E' \stackrel{\mathrm{df}}{=} C' \setminus C$ is concurrent to each event in $E'' \stackrel{\mathrm{df}}{=} C'' \setminus C$. Now, due to the consistency of Γ, no two distinct concurrent events in Γ^{Θ} can have the same signal label. Hence none of the events in E' can have the same signal label (even after ignoring the sign) as an event in E''. Consequently, $v^{E'} \odot v^{E''} = \mathbf{0}$. We next observe that $Code(C) + v^{E'} = Code(C') \geq Code(C'') = Code(C) + v^{E''}$ implies that $v^{E'} \geq v^{E''}$. Together with $v^{E'} \odot v^{E''} = \mathbf{0}$ and the fact that both $v^{E'} = v^{C'} - v^{C}$ and $v^{E''} = v^{C''} - v^{C}$ belong to $\{-1, 0, +1\}^k$, this leads to $v^{E'} \geq \mathbf{0} \geq v^{E''}$, and so $(*)$ holds.

[2] If the STG contains dummies then $Trig(e) = \max_{\prec} Strip(\langle e\rangle)$, where *Strip* is a function defined in Section 8.5.

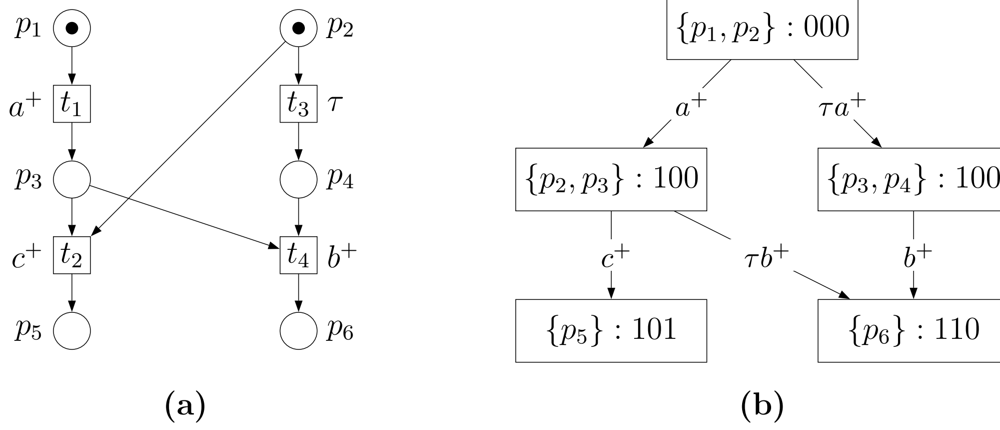

Figure 8.6: An STG **(a)** and its state graph **(b)** with dummies playing a confusing role.

By $(*)$, we immediately obtain that $Code(C') = Code(C'')$ implies $Code(C') = Code(C) = Code(C'')$. Moreover, if the states represented by C' and C'' are in USC (resp. CSC) conflict then their final markings (resp. sets of enabled output signals) are distinct, and thus the final marking (resp. set of enabled output signals) of C differs from at least one of them. Hence the proposition holds for USC and CSC conflicts.

Suppose now that s' and s'' are, without loss of generality, in p-normalcy conflict due to $Code(C') \geq Code(C'')$ and $Nxt_z(C') < Nxt_z(C'')$, which implies that $Nxt_z(C') = 0$ and $Nxt_z(C'') = 1$. Then, by $(*)$, if $Nxt_z(C) = 0$ then s and s'' are in p-normalcy conflict; otherwise, $Nxt_z(C) = 1$ and s and s' are in p-normalcy conflict. $\square$

In order to make the algorithm consider only ordered configurations, it is enough to choose an appropriate function $\hat{\xi}(x', x'') \stackrel{\text{df}}{=} (MCC(x'), MCC(x' \vee x''))$ (see Section 7.3), where '$\vee$' is the bitwise 'or' operation on binary vectors. Note that if we are verifying the normalcy property using this improvement, one cannot say in advance whether $C' \subset C''$ or $C'' \subset C'$ is the solution, and so the algorithm has to check both cases.

8.5 Handling dummy events

So far the proposed approach was applied to STGs without 'dummies', i.e., τ-labelled transitions, which are unrelated to any signal value changes.

The presence of dummy events may sometimes create problems in analyzing state coding conditions due to the inherent ambiguity involved in their interpretation. Indeed, the binary encodings of two states which are connected only by dummy transitions are the same and, formally, they are in USC conflict. But

from the technical point of view, dummies do not correspond to any physical transition made by a circuit, i.e., do not change the 'semantical' state of the system, and one should not interpret such a situation as a coding conflict. However, there are STGs for which the interpretation may be not entirely clear. For example, consider the STG together with a modified state graph shown in Figure 8.6(a,b), which has been obtained by applying the well-known procedure for eliminating all the 'silent' transitions, described for finite automatons in [48]. This procedure works on SG_Γ and produces the 'essential' edges that are associated with the 'transitions' of the form $s' \xrightarrow{\tau^* z^\pm} s''$. Such edges lead to a subset of states, which are the result of firing a non-dummy signal transition. Then it removes all τ-labelled transitions and the states which have become unreachable, together with their incoming and outgoing arcs.

This technique treats the markings $\{p_2, p_3\}$ and $\{p_3, p_4\}$ as not equivalent, e.g., because only the former enables a transition labelled by c^+. From another point of view, though, these two states are separated only by a dummy transition, and $\{p_3, p_4\}$ may be considered as a hidden intermediate state when τb^+ fires at $\{p_2, p_3\}$.

We outline another, *partial order* interpretation of dummies based on the branching processes of STGs. The idea is, given a configuration, to remove all its trailing τ-labelled events, yielding what we call an *essential* configuration. Other configurations can be seen as 'intermediate', and do not correspond to the 'real' states of the circuit. Therefore, one may check for coding conflicts involving only essential configurations. Formally, given a configuration of a branching process C, we denote by $\mathit{Strip}(C)$ the minimal configuration containing all its events labelled by signal transition labels. Clearly, $\mathit{Strip}(C)$ is uniquely defined. In Figure 8.6, the STG coincides with its unfolding, and the only essential configurations are $\emptyset$, $\{t_1\}, \{t_1, t_2\}$, and $\{t_1, t_3, t_4\}$.[3]

The modified algorithm will consider only the configurations of the form $\mathit{Strip}(C)$ as those which can be in coding conflict. Note that such a semantics has no obvious interpretation on the state graph level, since whether a state is essential or not may depend not only on the state itself, but also on the execution path through which it is reached, and on the causal relationships between the transitions involved in such a path. Indeed, in the unfolding there may be several configurations representing the same state s, but some of them have trailing dummies while the others have not. In such a case we say that s exhibits a *backward signal confusion*.

In order to retain only the essential solutions of the system of constraints described in Section 8.2, we can add the constraints

$$x(e) \leq \sum_{f \in (e^\bullet)^\bullet \setminus E_{cut}} x(f), \quad \text{for all } \tau\text{-labelled events } e \in E \setminus E_{cut} \ ,$$

requiring that if a dummy has fired then at least one of its causal descendants

[3] The initial event $\perp$ is not shown here.

has also fired, and so this dummy is not a trailing one. But this approach is not entirely satisfactory, since the number of constraints increases, and the algorithm still has to consider some of the 'intermediate' configurations just to find out that they are not solutions. A better way is to require all the maximal events of a configuration to be non-dummy. This can be easily achieved by restricting the condition in the header of the **for all** loop of the algorithm in Figure 7.6.

Another problem caused by the dummies is that computing the set of enabled outputs (needed for CSC and normalcy checking) becomes quite complicated. Indeed, one can compose an STG where it is possible to fire a long (even arbitrary long) sequence of dummies before any signal transition becomes enabled. Furthermore, in the finite prefix of the net unfolding, the correspondent sequence of events may lead beyond the cut-off events.

One solution is to (partially) unfold the STG again, starting from $Mark(C)$ and not generating events beyond any signal event. In order to minimize the number of times this is done, we can first check if the configurations are in USC conflict. Assuming that there are not many CSC conflicts which are not USC conflicts, the performance of the CSC checking algorithm should not suffer much. But for normalcy, the number of times this is done can still be quite large.

An alternative solution is to build the prefix beyond the cut-offs, so that any output signal event enabled (possibly, through a sequence of dummies) by any configuration containing no cut-off events can be reached. The algorithm does not have to assign new variables to these added events, it keeps them just for finding the sets of enabled outputs.

What we have described above works for all STGs without states exhibiting the backward signal confusion. In future work, we intend to clarify the semantics of such states.

It is important to note that Proposition 8.1 holds in the case of STGs containing dummies as well. In such a case, C' and C'' must be essential, $C \stackrel{\text{df}}{=} Strip(C' \cap C'')$, and the adduced proof can be adopted with minor modifications.

8.6 Experimental results

When testing the performance of the proposed algorithm we attempted the CSC conflict detection problem, i.e., non-linear systems of constraints were solved.

The results of the experiments are summarized in Tables 8.1–8.3. The meaning of the columns is as follows (from left to right): the name of the problem; the number of places, transitions, and signals in the original STG; the number of conditions, events and cut-off events in the complete prefix; the time spent by a special version of the PETRIFY tool, which did not attempt to resolve the coding conflicts it had identified; and the time spent by the new algorithm. We use 'time' to indicate that the test had not stopped after 15 hours. We have not included in the tables the time needed to build complete prefixes, since it did not exceed 0.1sec for all the attempted STGs. The STGs with names containing the

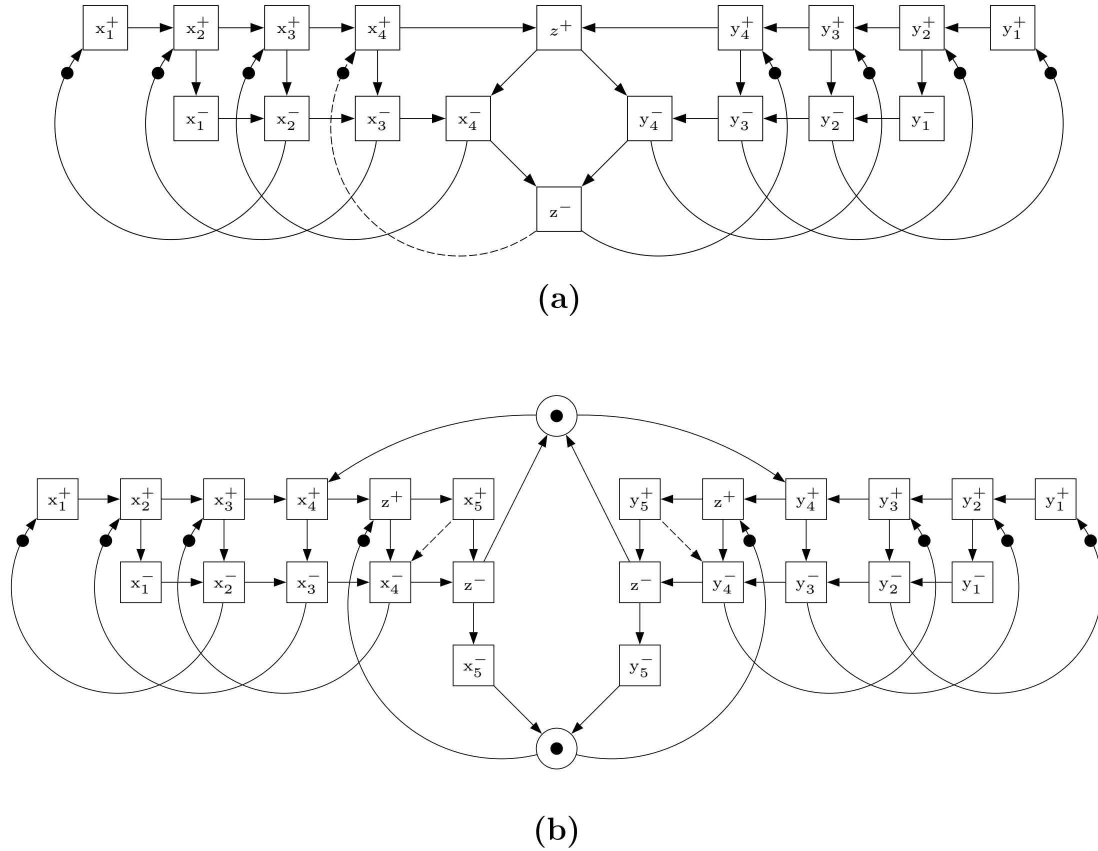

Figure 8.7: STG models of two weakly synchronized pipelines without arbitration **(a)** and with arbitration **(b)**.

occurrence of 'Csc' satisfy the CSC property, the others exhibit CSC conflicts.

We used several groups of benchmarks. The first group consisted of standard STG benchmarks, which are relatively small STGs coming from the academic practice of control circuit synthesis. We used them for testing the correctness of the implementation of the proposed algorithm, but have not included the timing results in the tables since all these examples were rather trivial. Another group of examples came from real design practice. They are as follows:

- LazyRing and Ring — Asynchronous Token Ring Adapters described in [10, 72].
- Dup4ph, Dup4phCsc, Dup4phMtr, Dup4phMtrCsc, DupMtrMod, DupMtrModUtg, and DupMtrModCsc — control circuits for the Power-Efficient Duplex Communication System described in [35].
- CfSymCscA, CfSymCscB, CfSymCscC, CfSymCscD, CfAsymCscA, and CfAsymCscB — control circuits for the Counterflow Pipeline Processor described in [93].

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Z\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	Petrify	Clp
LazyRing	35	32	11	87	66	5	2.02	<0.01
Ring	147	127	28	763	498	59	1498	0.01
Dup4ph	133	123	26	144	123	11	35.85	<0.01
Dup4phCsc	135	123	26	146	123	11	34.79	<0.01
Dup4phMtr	109	96	22	117	96	8	25.40	<0.01
Dup4phMtrCsc	114	105	26	122	105	8	25.26	0.02
DupMtrMod	129	100	21	199	132	10	222	<0.01
DupMtrModUtg	116	165	21	344	218	65	623	<0.01
DupMtrModCsc	152	115	27	228	149	13	286	<0.01
CfSymCscA	85	60	22	1341	720	56	357	107
CfSymCscB	55	32	16	160	71	6	15.27	0.06
CfSymCscC	59	36	18	286	137	10	33.24	2.30
CfSymCscD	45	28	14	120	54	6	8.57	0.01
CfAsymCscA	128	112	34	1808	1234	62	3988	2807
CfAsymCscB	128	112	32	1816	1238	62	6144	3280

Table 8.1: Experimental results: real-life STGs.

Some of these STGs, although built by hand, are quite large in size (see Table 8.1).

Two other groups, PpWk(m, n) and PpArb(m, n) (see Tables 8.2 and 8.3 respectively), contain scalable examples of STGs modelling m pipelines of size n weakly synchronized without arbitration (in PpWk(m, n)) and with arbitration (in PpArb(m, n)). The former offers the possibility of studying the effect of the optimization described in Section 8.4 (all STGs in the PpWk(m, n) series are marked graphs, and so are free from dynamic conflicts). Figure 8.7 illustrates these two types of STGs. The versions of these models without the dashed arcs exhibit coding conflicts; the dashed arrows are indicating the way how coding conflicts can be resolved by adding extra causality constraints to the specification. These benchmarks allowed us to test the algorithm on almost identical specifications, which did not contain coding conflicts.

Note that in all cases the size of the complete prefix was relatively small. This can be explained by the fact that STGs usually contain a lot of concurrency but rather few conflicts, and thus the prefixes are not much bigger then the STGs themselves. As a result, the memory requirements of the new algorithm are very moderate: recall that it uses just $O(|E|)$ memory besides that needed to store the prefix, which for all the examples shown in the tables means not more than just few kilobytes (in contrast, Petrify was repeatedly swapping pages to the disk for some of the examples due to the need to build the whole state spaces of the STGs).

Although the performed testing was limited in scope, we can draw some con-

clusions about the performance of the proposed algorithm. If a specification contained a coding conflict, the new algorithm in most cases was able to find it very quickly; on the other hand, specifications without coding conflicts were much harder to deal with. This is because in such a case the algorithm has to explore the full search space. In the worst case, this may result in exploring all pairs of configurations, which can be much bigger than the number of reachable states. However, the heuristics we described allowed the algorithm to considerably reduce the number of considered configurations, and it was quite competitive.

8.7 Conclusions

Experimental results indicate that the algorithm proposed in this chapter is not memory-demanding and in most cases time-efficient, though on some of the examples in Tables 8.2 and 8.3 its performance was not entirely satisfactory. It is worth emphasizing that the proposed approach overcomes the memory limitations of existing state-based methods, while still offering quite good performance.

In future work, we plan to incorporate more heuristics used by general-purpose solvers in order to reduce the search space. Also, the algorithm admits efficient parallelization (see Section 7.11), even for the distributed memory architecture. Additional speedups may be gained by using non-local corresponding configurations, as described in ([43]), for reducing the size of complete prefixes and thus the search space to be explored by the algorithm. Finally, as we already mentioned, we intend to extend the method to STGs with markings exhibiting backward signal confusion.

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Z\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	PETRIFY	CLP
PPWK(2,3)	23	14	7	41	23	1	0.55	<0.01
PPWK(2,6)	47	26	13	119	62	1	12.49	<0.01
PPWK(2,9)	71	38	19	233	119	1	103	<0.01
PPWK(2,12)	95	50	25	383	194	1	1140	0.01
PPWKCSC(2,3)	24	14	7	38	20	1	0.43	0.01
PPWKCSC(2,6)	48	26	13	110	56	1	10.73	0.18
PPWKCSC(2,9)	72	38	19	218	110	1	102	16.92
PPWKCSC(2,12)	96	50	25	362	182	1	4806	1411
PPWK(3,3)	34	20	10	63	35	1	2.50	<0.01
PPWK(3,6)	70	38	19	183	95	1	240	0.01
PPWK(3,9)	106	56	28	357	182	1	4952	0.01
PPWK(3,12)	142	74	37	585	296	1	*time*	0.01
PPWKCSC(3,3)	36	20	10	57	29	1	1.84	0.01
PPWKCSC(3,6)	72	38	19	165	83	1	108	14.10
PPWKCSC(3,9)	108	56	28	327	164	1	18282	11188
PPWKCSC(3,12)	144	74	37	543	272	1	*time*	*time*

Table 8.2: Experimental results: marked graphs.

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Z\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	PETRIFY	CLP
PPARB(2,3)	38	24	11	94	52	2	3.24	<0.01
PPARB(2,6)	62	36	17	202	106	2	37.77	0.47
PPARB(2,9)	86	48	23	346	178	2	884	41.11
PPARB(2,12)	110	60	29	526	268	2	5851	3818
PPARBCSC(2,3)	40	24	11	96	52	2	2.80	0.03
PPARBCSC(2,6)	64	36	17	204	106	2	45.01	2.76
PPARBCSC(2,9)	88	48	23	348	178	2	410	231
PPARBCSC(2,12)	112	60	29	528	268	2	*time*	19618
PPARB(3,3)	56	36	16	161	90	3	18.49	0.02
PPARB(3,6)	92	54	25	341	180	3	1042	19.71
PPARB(3,9)	128	72	34	575	297	3	8302	14903
PPARB(3,12)	164	90	43	863	441	3	*time*	*time*
PPARBCSC(3,3)	59	36	16	164	90	3	15.38	0.51
PPARBCSC(3,6)	95	54	25	344	180	3	450	348
PPARBCSC(3,9)	131	72	34	578	297	3	16300	*time*
PPARBCSC(3,12)	167	90	43	866	441	3	*time*	*time*

Table 8.3: Experimental results: STGs with arbitration.

Conclusions

In this thesis, we presented a framework for model checking based on prefixes of Petri net unfoldings. In Chapter 2, the theory of canonical prefixes is proposed. It provides a powerful tool for dealing with different variants of the unfolding technique, in a flexible and uniform way. A fundamental result of that chapter is the existence of 'special' canonical prefix of the unfolding of a Petri net, which can be defined without resorting to any algorithmic argument.

Chapters 4 and 5 address the issue of efficient construction of canonical prefixes. The former describes an efficient method of generating the possible extensions of a branching process — the most time-consuming part of unfolding algorithms. The latter proposes a new unfolding algorithm, which admits an efficient parallelization and allows for additional optimizations.

In Chapter 6, the theory of canonical prefixes presented in Chapter 2 and the unfolding algorithms developed in Chapters 4 and 5 are generalized to a class of high-level Petri nets. This is done by establishing an important relation between the branching processes of a high-level net and those of its low-level expansion, which allows for importing results proven for branching processes of low-level nets into the theory of high-level nets. The proposed approach is conservative in the sense that all the verification tools employing the traditional unfolding prefixes can be reused.

Chapter 7 develops a new integer programming technique for efficient model checking employing unfolding prefixes. The conducted experiments indicate that problems with more than a hundred of thousands of variables can be solved. The technique can be used, e.g., for detecting deadlocks in Petri nets, checking the mutual exclusion of places, and various kinds of reachability analysis. Moreover, in Chapter 8, this technique is applied to check the Unique State Coding (USC), Complete State Coding (CSC), and normalcy properties of Signal Transition Graphs (STGs), used for specifying asynchronous circuits.

As a result of this work, several model checking tools have been developed, viz. the PUNF tool (see [50]) for unfolding low-level and high-level Petri nets and STGs, and the CLP tool (see [51]) for model checking various properties of Petri nets and STGs. They are available for download from the following URL:

`http://www.cs.ncl.ac.uk/people/victor.khomenko/home.formal/tools/tools.html`

We believe that the results contained in this thesis, on one hand, will help to understand better the issues relating to prefixes of Petri net unfoldings, and,

on the other hand, will facilitate the design of efficient model checking tools employing them. Moreover, they will contribute to a long term goal of our research, which is the development of techniques and methods for the analysis and verification of concurrent reactive systems based on structures representing causality and concurrency in a system's behaviour, rather than on the global state-space view of it. And, as a consequence, *to make the unfoldings as usable as the state graphs*, in order to allow flexible goal-oriented model checking to be performed for complex concurrent reactive systems at early stages of their development, with greater efficiency than what could be achieved with state traversal techniques. The specific objectives are:

(a) To further improve the efficiency of unfolding algorithms.

(b) To extend the unfolding based techniques to new classes of Petri nets, including time and priority nets, and nets with inhibitor and/or read arcs. Such extensions would allow a further leap to more complicated system models, e.g., PROMELA or π–calculus.

(c) To improve the efficiency of verification of linear time temporal logic properties through the development of a suitable parallel model checker, and to investigate the applicability of unfoldings to model checking of a branching time temporal logic formulae.

(d) To extend the applicability of the unfolding based techniques in the area of VLSI circuits design.

This would make the unfolding approach a truly general method for dealing with a wide range of key application areas, including both software and hardware.

Bibliography

[1] F. Ajili and E. Contejean: Complete Solving of Linear Diophantine Equations and Inequations Without Adding Variables. Proc. of *1st International Conference on Principles and Practice of Constraint Programming (CP'1995)*, U. Montanari and F. Rossi (Eds.). Springer-Verlag, Lecture Notes in Computer Science 976 (1995) 1–17.

[2] F. Ajili and E. Contejean: Avoiding Slack Variables in the Solving of Linear Diophantine Equations and Inequations. *Theoretical Computer Science* 173(1) (1997) 183–208.

[3] E. Best, R. Devillers, and M. Koutny: *Petri Net Algebra*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag (2001).

[4] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz: A Class of Composable High Level Petri Nets. Proc. of *Application and Theory of Petri Nets (ATPN'1995)*, G. DeMichelis and M. Diaz (Eds.). Springer-Verlag, Lecture Notes in Computer Science 935 (1995) 103–120.

[5] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz: An M-net Semantics of $B(PN)^2$. Proc. of *International Workshop on Structures in Concurrency Theory (STRICT'1995)*, J. Desel (Ed.). Berlin (1995) 85–100.

[6] E. Best and B. Grahlmann: *PEP. Documentation and User Guide. Version 1.4*. Manual (1995).

[7] E. Best and B. Grahlmann: PEP — More Than a Petri Net Tool. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1996)*, T. Margaria and B. Steffen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 397–401.

[8] R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* C-35-8 (1986) 677–691.

[9] A. Bystrov, D. J. Kinniment, and A. Yakovlev: Priority Arbiters. Proc. of *6th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'2000)*, IEEE Computer Society Press (2000) 128–137.

[10] C. Carrion and A. Yakovlev: Design and Evaluation of Two Asynchronous Token Ring Adapters. Technical Report CS-TR-562, Department of Computing Science, University of Newcastle (1996).

[11] T.-A. Chu: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD Thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, MIT/LCS/TR-393 (1987).

[12] E. M. Clarke, O. Grumberg, and D. E. Long: Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems* 16(5) (1994) 1512–1542.

[13] E. M. Clarke, O. Grumberg, and D. Peled: *Model Checking*. MIT Press (1999).

[14] P. M. Cohn: *Algebra, volume 2*. John Wiley & sons (1977).

[15] P. M. Cohn: *Universal Algebra*. Reidel, 2nd edition (1981).

[16] E. Contejean: Solving Linear Diophantine Constraints Incrementally. Proc. of *10th International Conference on Logic Programming (ICLP'1993)*, D. S. Warren (Ed.). MIT Press (1993) 532–549.

[17] E. Contejean and H. Devie: Solving Systems of Linear Diophantine Equations. Proc. of *3rd Workshop on Unification*, University of Keiserlautern (1989).

[18] E. Contejean and H. Devie: An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations. *Information and Computation* 113 (1994) 143–172.

[19] J. C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering* 22(3) (1996) 161–180.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*. MIT Press, 2nd edition (2001).

[21] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems* E80-D(3) (1997) 315–325.

[22] J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno, and A. Yakovlev: Complete State Encoding Based on Theory of Regions. Proc. of *2nd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'1996)*, IEEE Computer Society Press (1996) 36–47.

[23] CPLEX Corporation: *CPLEX 3.0*. Manual (1995).

[24] J.-M. Couvreur, S. Grivet, and D. Poitrenaud: Unfolding of Products of Symmetrical Petri Nets. Proc. of *International Conference on Application and Theory of Petri Nets (ICATPN'2001)*, J.-M. Colom and M. Koutny (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2075 (2001) 121–143.

[25] M. Davis: *The Undecidable.* Raven Press (1965).

[26] J. Engelfriet: Branching Processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.

[27] A. Edelman: The Mathematics of the Pentium Division Bug. *SIAM Review* 39(1) (1997) 54–67.

[28] J. Esparza: Decidability and Complexity of Petri Net Problems — an Introduction. In: *Lectures on Petri Nets I: Basic Models*, W. Reisig and G. Rozenberg (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1491 (1998) 374–428.

[29] J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. of *International Conference on Concurrency Theory (CONCUR'1999)*, J. C. M. Baeten and S. Mauw (Eds.). Invited paper, Springer-Verlag, Lecture Notes in Computer Science 1664 (1999) 2–20.

[30] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1996)*, T. Margaria and B. Steffen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.

[31] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* 20(3) (2002) 285–310.

[32] J. Esparza and C. Schröter: Reachability Analysis Using Net Unfoldings. Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'2000)*, H. D. Burkhard, L. Czaja, A. Skowron, and P. Starke (Eds.). Informatik-Bericht 140(2). Humboldt-Universitat zu Berlin (2000) 255–270.

[33] H. Fleischhack, B. Grahlmann: A Petri Net Semantics for $B(PN)^2$ with Procedures which Allows Verification. Technical Report 21, Universität Hildesheim (1996).

[34] H. Fleischhack, B. Grahlmann: A Petri Net Semantics for $B(PN)^2$ with Procedures. Proc. of *2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'1997)*, IEEE Computer Society Press (1997) 15–27.

[35] S. B. Furber, A. Efthymiou, and M. Singh: A Power-Efficient Duplex Communication System. Proc. of *International Workshop on Asynchronous Interfaces: Tools, Techniques and Implementations (AINT'2000)*, A. Yakovlev and R. Nouta (Eds.). TU Delft, The Netherlands (2000) 145–150.

[36] J. D. Garside, S. B. Furber, and S.-H. Chung: AMULET-3 Revealed. Proc. of *5th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'1999)*, IEEE Computer Society Press (1999) 51–59.

[37] P. Godefroid: *Partial-Order Methods for the Verification of Concurrent Systems: an Approach to the State-Explosion Problem.* Springer-Verlag, Lecture Notes in Computer Science 1032 (1996).

[38] J. Goubault-Larrecq and I. MacKie: *Proof Theory and Automated Deduction.* Kluwer Academic Publishers (1997).

[39] K. Gödel: Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I [On Formally Undecidable Propositions of *Principia Mathematica* and Related Systems I]. *Monatshefte für Mathematik und Physik* 38 (1931) 173–198. Reprinted in [25, pp. 4–38].

[40] K. Heljanko: Deadlock Checking for Complete Finite Prefixes Using Logic Programs with Stable Model Semantics (Extended Abstract). Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'1998)*, Informatik-Bericht 110. Humboldt-Universitat zu Berlin (1998) 106–115.

[41] K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1999)*, Springer-Verlag, Lecture Notes in Computer Science 1579 (1999) 240–254.

[42] K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Technical Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo, Finland (1999).

[43] K. Heljanko: Minimizing Finite Complete Prefixes. Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'1999)*, Informatik-Bericht, Humboldt-Universitat zu Berlin (1999) 83–95.

[44] K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37(3) (1999) 247–268.

[45] K. Heljanko, V. Khomenko, and M. Koutny: Parallelization of the Petri Net Unfolding Algorithm. Technical Report CS-TR-733, Department of Computing Science, University of Newcastle (2001).

[46] K. Heljanko, V. Khomenko, and M. Koutny: Parallelization of the Petri Net Unfolding Algorithm. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2002)*, J.-P. Katoen and P. Stevens (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2280 (2002) 371–385.

[47] G. D. Holzmann: Software Analysis and Model Checking. Proc. of *International Conference on Computer Aided Verification (CAV'2002)*, E. Brinksma and K. G. Larsen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2404 (2002) 1–16.

[48] J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley (1979).

[49] K. Jensen: *Colored Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1992).

[50] V. Khomenko: PUNF *Documentation and User Guide. Version 6.01.* Manual (2002).

[51] V. Khomenko: CLP *Documentation and User Guide. Version 3.01β.* Manual (2002).

[52] V. Khomenko and M. Koutny: Deadlock Checking Using Liner Programming and Partial Order Dependencies. Technical Report CS-TR-695, Department of Computing Science, University of Newcastle (2000).

[53] V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. Technical Report CS-TR-711, Department of Computing Science, University of Newcastle (2000).

[54] V. Khomenko and M. Koutny: LP Deadlock Checking Using Partial Order Dependencies. Proc. of *International Conference on Concurrency Theory (CONCUR'2000)*, C. Palamidessi (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1877 (2000) 410–425.

[55] V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. *Formal Methods in System Design* (2002) submitted paper.

[56] V. Khomenko and M. Koutny: An Efficient Algorithm for Unfolding Petri Nets. Technical Report CS-TR-726, Department of Computing Science, University of Newcastle (2001).

[57] V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. Proc. of *International Conference on Concurrency Theory*

(CONCUR'2001), P. G. Larsen and M. Nielsen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2154 (2001) 366–380.

[58] V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. Technical Report CS-TR-763, Department of Computing Science, University of Newcastle (2002).

[59] V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, H. Garavel and J. Hatcliff (Eds.). Springer-Verlag, Lecture Notes in Computer Science (2003) to appear.

[60] V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Technical Report CS-TR-741, Department of Computing Science, University of Newcastle (2001).

[61] V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Proc. of *International Conference on Computer Aided Verification (CAV'2002)*, E. Brinksma and K. G. Larsen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2404 (2002) 582–595.

[62] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Technical Report CS-TR-736, Department of Computing Science, University of Newcastle (2001).

[63] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *International Conference on Design, Automation and Test in Europe (DATE'2002)*, C. D. Kloos and J. Franca (Eds.). IEEE Computer Society Press (2002) 338–345.

[64] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev: Checking Signal Transition Graph Implementability by Symbolic BDD Traversal. Proc. of *International Conference on Design, Automation and Test in Europe (DATE'1995)*, IEEE Computer Society Press (1995) 325–332.

[65] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev: Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings. Proc. of *International Conference on Application of Concurrency to Sysytem Design (ICACSD'98)*, IEEE Computer Society Press (1998) 152–163.

[66] D. König: Über eine Schlußweise aus dem Endlichen ins Unendliche. *Acta Litt. ac. sci. Szeged* 3 (1927) 121–130. Bibliography in: Theorie der endlichen und unendlichen Graphen. Teubner, Leipzig (1936, reprinted 1986).

[67] M. Koutny and E. Best: Fundamental Study: Operational and Denotational Semantics for the Box Algebra. *Theoretical Computer Science* 211 (1999) 1–83.

[68] V. E. Kozura: Unfolding of Colored Petri Nets. Technical Report 80, A. P. Ershov Institute of Informatics Systems (2000).

[69] S. Krivoy: About Some Methods of Solving and a Feasibility Criteria of Linear Diophantine Equations over the Natural Numbers Domain (in Russian). *Cybernetics and System Analysis* 4 (1999) 12–36.

[70] L. Lamport: A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems* 5(1) (1987) 1–11.

[71] R. Langerak and E. Brinksma: A Complete Finite Prefix for Process Algebra. Proc. of *International Conference on Computer Aided Verification (CAV'1999)*, N. Halbwachs and D. Peled (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1633 (1999) 184–195.

[72] K. S. Low and A. Yakovlev: Token Ring Arbiters: an Exercise in Asynchronous Logic Design with Petri Nets. Technical Report CS-TR-537, Department of Computing Science, University of Newcastle (1995).

[73] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. Proc. of *International Conference on Design, Automation and Test in Europe (DATE'2003)*, IEEE Computer Society Press (2003) to appear.

[74] K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *International Conference on Computer Aided Verification (CAV'1992)*, G. von Bochmann and D. K. Probst (Eds.). Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.

[75] K. L. McMillan: *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, CMU-CS-92-131 (1992).

[76] S. Melzer: *Verifikation Verteilter Systeme mit Linearer — und Constraint-Programmierung*. PhD Thesis. Technische Universität München, Utz Verlag (1998).

[77] S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *International Conference on Computer Aided Verification (CAV'1997)*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363.

[78] T. Murata: Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE* 77(4) (1989) 541–580.

[79] I. Niemelä and P. Simons: SMODELS — An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. Proc. of *4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'1997)*, Springer-Verlag, Lecture Notes in Artificial Intelligence 1265 (1997) 420–429.

[80] M. A. Peña and J. Cortadella: Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits. Proc. of *2nd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'1996)*, IEEE Computer Society Press (1996) 222–232.

[81] A. Pnueli: The Temporal Logic of Programs. Proc. of *18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press (1977) 46–57.

[82] A. Pnueli: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. Proc. of *Advanced School on Current Trends in Concurrency*, J. de Bakker, W.-P. de Roever, and G. Rozenberg (Eds.). Springer-Verlag, Lecture Notes in Computer Science 224 (1994) 510–584.

[83] E. Post: Recursively Enumearable Sets of Positive Integers and Their Decision Problems. *American Mathematical Society Bulletin* 50 (1944) 284–316. Reprinted in [25, pp. 304–337].

[84] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli: Binary Decision Diagrams on Network of Workstation. Proc. of *International Conference on Computer Design (ICCD'1996)*, IEEE Computer Society Press (1996) 358–364.

[85] S. Römer: *Entwicklung und Implementierung von Verifikationstechniken auf der Basis von Netzentfaltungen*. PhD thesis, Technische Universitat Munchen (2000).

[86] A. Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfoldings*. PhD Thesis, University of Newcastle upon Tyne (1997).

[87] A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno: Partial Order Approach to Synthesis of Speed-Independent Circuits. Proc. of *3rd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'1997)*, IEEE Computer Society Press (1997) 254–265.

[88] M. Silva, E. Teruel, and J.-M. Colom: Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems. In:

Lectures on Petri Nets I: Basic Models, W. Reisig and G. Rozenberg (Eds.). Springer-Verlag (1998) 309–373.

[89] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov, and A. Smirnov: Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs. Proc. of *International Conference on Application of Concurrency to Sysytem Design (ICACSD'2001)*, IEEE Computer Society Press (2001) 179–188.

[90] A. Turing: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2(42) (1936) 230–265. Reprinted in [25, pp. 115-154].

[91] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man: Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. Proc. of *International Conference on Computer-Aided Design (ICCAD'1990)*, IEEE Computer Society Press (1990) 184–187.

[92] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. *Formal Methods in System Design* 9(3) (1996) 139–188.

[93] A. Yakovlev: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets. *Formal Methods in System Design* 12(1) (1998) 39–71.

Index

(π, e)-extension, 8
Σ-compatible, 5
$\lhd$-chain, 2
$\prec$-chain, 6
ξ-minimal solution, 89
j-th basic default vector, 79
k-bounded M-net system, 66
k-bounded net system, 3
t-labelled, 6
Σ^{Θ}-compatible closure, 86

adequate order, 14
arcs, 115

backward signal confusion, 124
basic algorithm, 23
bounded M-net system, 66
bounded net system, 3
branching condition, 79, 83, 89, 93, 105
branching process, 6, 69, 116

canonical prefix, 19
cardinality of a multiset, 1
causal predecessors, 8
causality relation, 6
CDA, 79
cluster, 60
co-set, 8
complete branching process, 16
Complete State Coding (CSC), 116
concurrent, 6
conditions, 6
configuration, 8
conflict, 5
consistent STG, 115
Contejean and Devie's algorithm, 79
corresponding configuration, 18
coverable marking, 3
CSC conflict, 116
cut, 8
cut-off events, 18
cutting context, 14

dead transition, 3
deadlock-free, 3
deadlocked marking, 3
dense cutting context, 15
depth of an event, 6
descending $\lhd$-chain, 2
difference of two multisets, 1
downward-closed, 8

empty multiset, 1
enabled transition, 3
events, 6
expansion of an M-net, 68
extension, 8

feasible events, 18
final marking, 8
finite multiset, 1
firing mode, 65
firing of a transition, 3
flow relation, 2
frozen components, 80, 82, 91, 92

guard of a transition, 65

homomorphism from an occurrence net to a net system, 6
homomorphism from an occurrence net to an M-net system, 69

incidence matrix, 4
induces, 8
initial marking, 3, 66

initial state, 115
interleaving semantics, vii
intersection of two multisets, 1

labelling function, 114
legal firings, 65
legal place instances, 65
local configuration, 8

M-net, 65
M-net system, 66
marking, 2, 66
marking change vector, 60
marking equation, 5
minimal Σ^{Θ}-compatible closure, 86
multiset, 1
mutually exclusive, 3

n-normalcy, 120
negative normalcy, 120
net (with weighted arcs), 2
net system, 3
next-state function, 120
Noetherian induction, 2
normal STG, 120

occurrence net, 6
ordinary net, 2

p-normalcy, 120
Parikh vector, 5
place, 2, 65
positive normalcy, 120
possible extension, 8, 70
postset, 3
prefix of a branching process, 6
preset, 3
preset tree, 38

reachable markings, 3, 66
reachable states, 115
reactive systems, vii
represented, 8

safe M-net system, 66
safe net system, 3
saturated cutting context, 15
self-conflict, 6
separated, 6
separating constraint, 119
signal change vector, 114, 116
Signal Transition Graph (STG), 114
signal transition labels, 114
signals, 114
slack variables, 83
slice, 24
slicing algorithm, 24
speed-independent, 120
state assignment function, 115
state graph, 115
state of an STG, 114
state space explosion, v, viii
static cut-off events, 18
strictly k-bounded M-net system, 66
strictly safe M-net system, 66
suffix, 8
sum of two multisets, 1

transition, 2, 65
transition relation, 66
triggers, 121
true concurrency semantics, vii
type of a place, 65

unfolding, 6
Unique State Coding (USC), 115
USC conflict, 115

vector of initial signal values, 114

weight function, 2
weight of a preset tree, 39
well-founded order, 2