Helsinki University of Technology Laboratory for Theoretical Computer Science

Research Reports 71

Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 71

Espoo 2002 HUT-TCS-A71

COMBINING SYMBOLIC AND PARTIAL ORDER METHODS FOR MODEL CHECKING 1-SAFE PETRI NETS

Keijo Heljanko

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 71
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 71
Espoo 2002 HUT-TCS-A71

COMBINING SYMBOLIC AND PARTIAL ORDER METHODS FOR MODEL CHECKING 1-SAFE PETRI NETS

Keijo Heljanko

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium T2 at Helsinki University of Technology (HUT CS building, Espoo, Finland) on the 22nd of March, 2002, at 12 noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-5886-2
ISSN 1457-7615

Picaset Oy
Helsinki 2002

ABSTRACT: In this work, methods are presented for model checking finite state asynchronous systems, more specifically 1-safe Petri nets, with the aim of alleviating the state explosion problem. Symbolic model checking techniques are used, combined with two partial order semantics known as net unfoldings and processes.

We start with net unfoldings and study deadlock and reachability checking problems, using complete finite prefixes of net unfoldings introduced by McMillan. It is shown how these problems can be translated compactly into the problem of finding a stable model of a logic program. This combined with an efficient procedure for finding stable models of a logic program, the Smodels system, provides the basis of a prefix based model checking procedure for deadlock and reachability properties, which is competitive with previously published procedures using prefixes.

This work shows that, if the only thing one can assume from a prefix is that it is complete, nested reachability properties are relatively hard. Namely, for several widely used temporal logics which can express a violation of a certain fixed safety property, model checking is PSPACE-complete in the size of the complete finite prefix.

A model checking approach is devised for the linear temporal logic LTL-X using complete finite prefixes. The approach makes the complete finite prefix generation formula specific, and the prefix completeness notion application specific. Using these ideas, an LTL-X model checker has been implemented as a variant of a prefix generation algorithm.

The use of bounded model checking for asynchronous systems is studied. A method to express the process semantics of a 1-safe Petri net in symbolic form as a set of satisfying truth assignments of a constrained Boolean circuit is presented. In the experiments the BCSat system is used as a circuit satisfiability checker. Another contribution employs logic programs with stable model semantics to develop a new linear size bounded LTL-X model checking translation that can be used with step semantics of 1-safe Petri nets.

KEYWORDS: Verification, Model Checking, Petri nets, Complete Finite Prefixes, Partial Order Methods, Symbolic Methods, Bounded Model Checking

CONTENTS

PREFACE

This dissertation is the result of studies and research at the Laboratory for Theoretical Computer Science of Helsinki University of Technology from 1997 to 2002. I'm grateful to my supervisor Prof. Ilkka Niemelä, for frequent advice and great support. I'm also grateful to Prof. Emeritus Leo Ojala who deserves a large credit for supervising me until his retirement in the year 2000, and for creating a good research oriented laboratory.

People at the laboratory deserve credit for a good research atmosphere. A list of people to thank would be too long, however, I would like to especially thank Tommi Junttila for his comments on numerous research ideas and issues.

During 1999 and 2000 I visited Prof. Javier Esparza's research group at Technische Universität München for a total of 8 months. I would like to thank for the visit opportunity, as these visits were vital for creating a major part of this dissertation.

My co-authors J. Esparza and I. Niemelä significantly contributed to the joint publications and deserve credit for their excellent work. They also gave me a much needed insight into their fields of expertise.

The following people directly contributed software or examples used in the experiments of this dissertation: Patrik Simons (Smodels), Tommi Junttila (BCSat), Gerard Holzmann (SPIN tool LTL to Büchi automata translator [44]), Frank Wallner (qq tool for synchronizing Petri nets and Büchi automata), Stefan Schwoon (qq tool support), Burkhard Graves and Bernd Grahlmann (C code to read PEP prefix files), Stefan Römer (ERVunfold binaries and example nets), Stephan Melzer (example nets), Claus Schröter (example nets and formulas). I would also like to thank Victor Khomenko for interesting discussions on the net unfolding method.

The work was funded by Helsinki Graduate School on Computer Science and Engineering (HeCSE) and by the Academy of Finland (projects 43963 and 47754). The financial support of the following institutions is gratefully acknowledged: Support Foundation of Helsinki University of Technology, Emil Aaltonen Foundation, Nokia Oyj Foundation, Helsinki University of Technology grant fund, and Foundation of Technology (Tekniikan Edistämissäätiö). The grants of these institutions were of great importance as they made full-time studies and international visits possible.

I would like to thank my parents for their support and encouragement. Last but not least I would like to thank my love Virpi for her love and support.

Otaniemi, February 28th, 2002

Keijo Heljanko

LIST OF PUBLICATIONS

The dissertation consists of 6 publications listed below, and a dissertation summary. Publications [**P1**]-[**P4**] are on model checking using complete finite prefixes, and [**P5**]-[**P6**] are on bounded model checking.

[**P1**] K. Heljanko, Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets, *Fundamenta Informaticae*, 37(3):247–268, 1999, IOS Press.

[**P2**] K. Heljanko, Model checking with finite complete prefixes is PSPACE-complete, in *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'2000)*, State College, Pennsylvania, USA, August 2000, volume 1877 of Lecture Notes in Computer Science, pages 108–122, Springer-Verlag.

[**P3**] J. Esparza and K. Heljanko, A new unfolding approach to LTL model checking, in *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, Geneva, Switzerland, July 2000, volume 1853 of Lecture Notes in Computer Science, pages 475–486, Springer-Verlag.

[**P4**] J. Esparza and K. Heljanko, Implementing LTL model checking with net unfoldings, in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, Toronto, Canada, May 2001, volume 2057 of Lecture Notes in Computer Science, pages 37–56, Springer-Verlag.

[**P5**] K. Heljanko, Bounded reachability checking with process semantics, in *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, Aalborg, Denmark, August 2001, volume 2154 of Lecture Notes in Computer Science, pages 218–232, Springer-Verlag.

[**P6**] K. Heljanko and I. Niemelä, Bounded LTL model checking with stable models, in *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, Vienna, Austria, September 2001. volume 2173 of Lecture Notes in Artificial Intelligence, pages 200–212, Springer-Verlag.

The current author is the only author of publications [**P1**], [**P2**], and [**P5**].

The publication [**P3**] (as well as the extended version [20]) is co-authored by J. Esparza. The key ideas of the publication [**P3**] were jointly developed while the current author was visiting Technische Universität München. Some key proof ideas on the tableaux side of [20] are due to J. Esparza, whereas the proofs of the used synchronization construction are by current author.

The publication [**P4**] (as well as the extended version [22]) is co-authored by J. Esparza. The paper was jointly written, with the current author designing the presented algorithms, implementing the unfsmodels prototype tool, and performing the experiments. J. Esparza contributed to simplifying the theory behind the used approach, as well as coming up with simpler proofs in [22].

The publication [**P6**] is co-authored by I. Niemelä. The basic translation was jointly developed. The current author's contribution is a new succinct LTL translation, implementation of the translations, and the experimental work.

1 INTRODUCTION

It is a widely recognized fact that the complexity of systems containing software or hardware components is increasing at a high rate. Furthermore, we as a society are increasingly more dependent on e.g., the correct functioning of communication infrastructure. Therefore, computer scientists are faced with the problem of designing safety or business critical systems of large complexity.

The traditional way of ensuring the correctness of of such systems has relied on two main techniques of *testing* and *simulation*. However, when the systems contain concurrent and/or reactive components, these techniques frequently do not scale at the rate of the system complexity growth. The use of *computer aided verification* has been suggested as an aid to supplement these methods. These methods are based on the observation that we should use the increasing computational capacity we have at our disposal to ease the design of complex systems.

One of the most promising methods for computer aided verification is *model checking*. The basic principles of model checking were developed in the early 1980's independently by two different groups with the earliest publications being [12, 64]. The basic idea is to model the system of interest so as to allow the generation of a graph that contains the reachable states of the system as nodes and the state transitions between them as edges. When a labeling of the nodes with *atomic propositions* which hold at each state is added, this graph is known as a *Kripke structure* of the system (see e.g., [11]). The specification of the property we are interested in is given by a *temporal logic formula*. After this, one can check with a *model checking algorithm* whether the system meets its specification, i.e., by checking if the Kripke structure of the system is a model of the specification.

The system under model checking can be described in several different ways. In case of synchronous digital hardware the description language is frequently a hardware description language. In the case of asynchronous systems, there is a variety of different formalisms. In this work, a model called (1-safe) Petri nets [15] will be used as a system description formalism. Other choices of asynchronous system models could include process algebras [43, 57], extended finite state machines as supported by the SPIN tool [44], high-level Petri nets as supported by tools like Maria and PROD [53, 73], protocol specification languages such as SDL [45], and various programming languages such as Java [2]. We selected 1-safe Petri nets mainly because of their simplicity and the fact that there is an extensive body of research on their analysis methods. We believe that most of the methods developed in this work can also be applied to other asynchronous system description formalisms.

The use of temporal logics to specify properties of concurrent programs was first suggested by Pnueli in the late 1970's [63]. Several temporal logics can be used as specification languages. Two of the most prominent are the computation tree logic (CTL), and linear temporal logic (LTL); for the semantics of the logics and a discussion of their features see e.g., [11]. This work will concentrate mainly on LTL, particularly on its subset LTL-X, in which the so called *next-time* operator has been removed. The LTL-X logic is probably the most widely used linear time temporal logic used to specify properties of asynchronous systems.

After the system model is described, and the specifications are developed, model checking is (ideally at least) a fully automated procedure. A model checker will either output that the system corresponds to the specification or that the specified property does not hold. The *executions* of a system are the finite and infinite paths in the Kripke structure that start from some initial state of the system. If the specified property does not hold, the model checker outputs a *counterexample* execution which violates the property. This frequently facilitates in the location of errors. These features have made model checking an appealing alternative for industrial use.

The main obstacle in applying model checking is the *state explosion problem*. For example, if the system is described as a composition of n finite state machines, then the Kripke structure of the system can be of exponential size in the number of components. Some sources of state explosion are concurrency of components mentioned above, and combinatorial explosion due to combinations of different data values in data variables.

In this work, methods for alleviating the state explosion problem in model checking of 1-safe Petri nets are developed. The two main techniques employed in this work are the use of symbolic and partial order methods. There is a large body of work dedicated to making model checking more efficient in different domains; for an overview see e.g., [11, 70].

In symbolic model checking the main idea is to represent the behavior of the system in a symbolic form rather than explicitly constructing a Kripke structure as a graph. There are several variations to symbolic methods. Their common feature is the use of representations of sets of states of the system in implicit form rather than having each global state of a system explicitly represented, e.g., as a node of the Kripke structure.

There are a large number of symbolic methods available. The most well-known is the use of data structure called ordered binary decision diagrams (OBDDs), which are a canonical form of Boolean functions [9]. The method was developed by McMillan for the verification of synchronous digital circuits [10, 55]. The main idea is to represent the transition relation and sets of reachable states as Boolean functions represented by OBDDs. The OBDDs have efficient algorithms for basic Boolean operations, and are frequently very compact exploiting regularities in the digital circuits to represent large sets of states of synchronous hardware designs very compactly.

Recently, several suggestions have been made to replace OBDDs with methods based on propositional satisfiability (SAT) procedures [1, 6] to further improve the scalability of symbolic model checking. The *bounded model checking* method was introduced in [6]. The main idea in bounded model checking is to look for counterexamples that are shorter than some fixed length n for a given property. If a counterexample can be found which is at most of length n, the property does not hold for the system. Otherwise the result is inconclusive, and the bound must be increased or proved sufficient to cover all possible counterexamples by other means. The implementation ideas are very similar to procedures used in SAT-based artificial intelligence (AI) planning [47, 59].

It seems that the bounded model checking procedures can currently challenge OBDD based methods on digital hardware designs both in terms of memory and time required to find counterexamples [7, 8, 13]. The weakness of bounded model checking is that if no counterexample can be found using a bound, the result is in general inconclusive. In certain favorable cases, it can be proved that e.g., all reachable states of a system are reachable within some bound n. In this case the bounded model checking is able to show the non-existence of counterexamples for reachability properties; for more discussion see [6]. Another way of ensuring completeness is to use a SAT procedure inside a "classical" symbolic model checker, as presented in [1, 76], to replace an OBDD based procedure. It has also been observed that the performance of a SAT based procedure is frequently more predictable than the performance of an OBDD based procedure, which depends on the so called *variable ordering* that can dramatically affect the OBDD sizes during model checking, see e.g., [13]. This variable ordering is frequently hard to generate automatically using heuristics, and generally more human interaction is needed when using OBDDs instead of SAT procedures [13].

Partial order methods are a collection of methods to alleviate the state explosion problem during the verification of asynchronous systems. Most of the communication protocol and software verification work has used some kind of asynchronous system model, in which there are a set of different "modules" in the system that can each operate independently of each other without a global synchronization clock. The modules can communicate with each other through such communication mechanisms as shared variables, message queues, or synchronization primitives. In partial order methods the goal is to use the independence between the modules of the system to alleviate the state explosion problem. This independence arises from the fact that frequently two modules of the system do not interact with each other and e.g., changes made to local variables in two different modules of the system can frequently be executed in any order (or event concurrently) without affecting the outcome (the reached global state of the system). The partial order methods can be divided into two subclasses: *partial order reduction* and *partial order semantics* methods.

The first class of partial order methods includes the so called partial order reduction methods, which use the independence between transitions of the system to generate a subset of the Kripke structure of the system, which still preserves the model checking outcome of the specification we are interested in. The methods thus use the independence information to prune the Kripke structure, but still operate on interleaving executions of the system. As noted in Section 10 of [11] this set of methods could be better described as *model checking using representatives*, since the verification is performed using representative executions from suitably defined (using a notion of independence) equivalence classes of behaviors.[1] Methods of this class include *stubborn sets*, *persistent sets*, and *ample sets*. Also closely related to these methods is the *sleep set* method. For more information on this class of methods, see e.g., [11, 30, 70, 72].

This work will concentrate on the second subclass of partial order methods

[1]Rather than being based on a partial order model of program execution, the methods use commutativity of (some) transitions to generate only part of the Kripke structure. This commutativity might not even arise from concurrency.

which will be referred to as partial order semantics methods. Namely, these methods take a partial order view of the behavior of asynchronous systems. In this view, two concurrent (and thus also independent) events are executed concurrently, and not in any fixed order. There is an order between events that are dependent on each other; however, as noted before, this order need not be a total order, but only a partial one. For Petri nets, there are two main partial order semantics, known as *net unfoldings* and *processes*. Roughly speaking, a net unfolding can be seen as a partial order branching time model of computation, while processes are a linear time view of the partial order behavior of the system.

Net unfoldings were introduced in [58] as a partial order semantics for Petri nets. Intuitively, they can be seen as a partial order version of an infinite computation tree. They were later more extensively researched under the name of branching processes by Engelfriet [16]. McMillan was the first to show how to use net unfoldings as a basis for a verification method [55] for finite state systems. He provided an algorithm to compute a *complete finite prefix* of the unfolding, which contained full information about the behavior of the Petri net system in symbolic form. The finite prefix can sometimes be exponentially more succinct than the Kripke structure of the system, which makes them interesting symbolic representations for model checking work. The net unfoldings have been a base of several model checking approaches in the past, some of which are mentioned in this work. Section 3 contains more information on the net unfolding method. In addition, publications [**P1**]-[**P4**] are on net unfolding based methods.

Processes were introduced to give partial order semantics to Petri nets [4, 5, 29, 31]. Intuitively, a process can be seen as the partial order version of an execution. However, it is actually the case that even a single process can correspond to exponentially many interleaving executions, and thus can sometimes also be exponentially more succinct than the Kripke structure. The partial order behavior of the system can now be described as a set of processes it induces. A net unfolding can be intuitively seen as the union of all the processes of the Petri net system, with maximal prefix sharing. The term "branching processes" refers to this fact. The only process based verification method we know of is presented in publication [**P5**], which contains a process based bounded model checking procedure. It can be seen as a symbolic representation of all the processes of the 1-safe Petri net in question, which have a depth equal to a user specified value n. An extensive treatment of Petri net processes can be found in [4, 5].

For computational complexity of Petri net related verification problems, see e.g., [18]. More information on the state explosion problem and methods to alleviate it can be found in [70]. For a longer introduction to model checking, including other related techniques, see e.g., [11].

Our research goal has been the development of efficient model checking methods for 1-safe Petri nets. The main problem facing us is the state explosion problem. We have chosen to concentrate on using a combination of symbolic methods and partial order semantics to alleviate this problem. A major part of the work focuses on developing a better understanding of complete finite prefix based verification methods.

1.1 Contributions

The main contributions of each of the publications are the following:

- [**P1**]: Linear size translations are devised from the deadlock and reachability problems of 1-safe Petri nets using complete finite prefixes into finding a stable model of a logic program. For deadlock checking, the translation can be seen as an adaptation of the mixed integer programming translation of [56] to the used logic programming framework, while the reachability checking version is new. Experimental results from the deadlock detection problem show the method to be competitive with alternative net unfolding based deadlock checking approaches. This publication is an extended version of [37].

- [**P2**]: Model checking several temporal logics is shown to be PSPACE-complete in the size of a complete finite prefix of a 1-safe Petri net system for fixed size formulas. The proof employs a class of net systems for which it is easy to generate a complete finite prefix in polynomial time. This class is also shown to contain net systems for which classical prefix generation algorithms [24, 25, 55] generate exponentially larger prefixes than required to satisfy the prefix completeness criterion.

- [**P3**]: A new net unfolding based model checking procedure for an action based linear temporal logic is presented. This procedure solves the model checking problem by direct inspection of a prefix instead of requiring the running of an elaborate algorithm on the prefix, as is the case in previous approaches [75]. The report version [20] is an extended version that contains proofs and examples not contained in the conference version [**P3**] due to length constraints.

- [**P4**]: An implementation of a net unfolding based linear temporal logic model checker is presented. The tableau procedure of [**P3**] is applied to a state based temporal logic LTL-X, and developed further to be more easily implementable as a modification of a conventional prefix generation procedure. Experimental results from a prototype implementation are presented. Again, the report version [22] contains proofs omitted from the conference version [**P4**] due to length constraints.

- [**P5**]: Bounded model checking is applied to checking reachability properties of asynchronous systems, specifically 1-safe Petri nets. We consider three different semantics: interleaving, step, and process semantics. The reachability checking problems are translated into constrained Boolean circuit satisfiability, and experimental results on a set of deadlock checking problems are obtained. The main contribution is the translation for the process semantics, which frequently performs best of the three semantics considered.

- [**P6**]: We present how to use logic programs with stable model semantics to solve bounded model checking problems of 1-safe Petri nets.

In this work, two semantics are considered: interleaving and step semantics. As properties, reachability and also linear time temporal logic LTL-X are used, both with parametric initial markings. The main contribution of the paper is a new, more succinct bounded LTL-X model checking translation that allows for concurrency of invisible transitions in generated counterexamples. This frequently allows counterexamples to be found with smaller bounds. This publication is an extended version of [41].

Structure of the Dissertation. The dissertations consists of 6 publications and a dissertation summary.

The structure of the dissertation summary is as follows. First we introduce basic notation used for Petri nets. In Section 3 we define net unfoldings. The discussion of different notions of prefix completeness in Section 3.1 is a subject that is presented only in the dissertation summary. We will continue with the definition of logic programs with stable model semantics, and the motivation for using them in Section 4. Verification with prefixes is the topic of Section 5, parts of which are new to this work. In Section 6, we introduce bounded model checking. The conclusions are in Section 7. The Appendix A contains corrections and additions to the publications.

2 PETRI NETS

This section summarizes the basic Petri net notation used throughout the work. All of the material is also presented in the publications.

Petri nets are a widely used model of concurrent and reactive systems. In this work we discuss verification methods for Petri nets used to model asynchronous finite state systems. More specifically, we limit ourselves to the so-called 1-safe Petri nets, which can be seen as an interesting generalization of communicating automata, see e.g., [15].

A *net* is a triple $N = \langle P, T, F \rangle$, where P and T are disjoint sets of *places* and *transitions*, respectively, and F is a function $(P \times T) \cup (T \times P) \rightarrow \{0, 1\}$. Places and transitions are generically called *nodes*. If $F(x, y) = 1$ then we say that there is an *arc* from x to y. The places are represented in graphical notation by circles, transitions by squares, and the *flow relation* F with arcs.

The *preset* of a node $x \in P \cup T$, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. The *postset* of a node $x \in P \cup T$, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. Their generalizations on sets of nodes $X \subseteq P \cup T$ are defined as ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$, and $X^\bullet = \bigcup_{x \in X} x^\bullet$, respectively. In this work we consider only nets in which every transition has a nonempty preset *and* a nonempty postset.

A *marking* of a net $\langle P, T, F \rangle$ is a mapping $P \rightarrow I\!N$ (where $I\!N$ denotes the set of natural numbers including 0). We identify a marking M with the multiset containing $M(p)$ copies of p for every $p \in P$. For instance, if $P = \{p_1, p_2\}$ and $M(p_1) = 1$, $M(p_2) = 2$, we write $M = \{p_1, p_2, p_2\}$. A marking is graphically denoted by a distribution of tokens on the places of the net.

A marking M *enables* a transition t if it marks each place $p \in {}^\bullet t$ with a token, i.e., if $M(p) > 0$ for each $p \in {}^\bullet t$. If t is enabled at M, then it can *fire* or *occur*, and its occurrence *leads to* a new marking M', obtained by removing a token from each place in the preset of t, and adding a token to each place in its postset; formally, $M'(p) = M(p) - F(p, t) + F(t, p)$ for every place p. For each transition t the relation $\xrightarrow{t}$ is defined as follows: $M \xrightarrow{t} M'$ if t is enabled at M and its occurrence leads to M'.

A 4-tuple $\Sigma = \langle P, T, F, M_0 \rangle$ is a *net system* if $\langle P, T, F \rangle$ is a net and M_0 is a marking of $\langle P, T, F \rangle$ (called the *initial marking* of Σ). We will use as a running example the net system in Figure 1.

A sequence of transitions $\sigma = t_1 t_2 \ldots t_n$ is an *occurrence sequence* if there exist markings $M_1, M_2, \ldots, M_n$ such that

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots M_{n-1} \xrightarrow{t_n} M_n.$$

M_n is the marking reached by the occurrence of σ, which is also denoted by $M_0 \xrightarrow{\sigma} M_n$. A marking M is a *reachable marking* if there exists an occurrence sequence σ such that $M_0 \xrightarrow{\sigma} M$. The *reachability graph* of a net system Σ is the labelled graph having the reachable markings of Σ as nodes, and the $\xrightarrow{t}$ relations (more precisely, their restriction to the set of reachable markings) as edges. In this work we only consider net systems with finite reachability graphs.

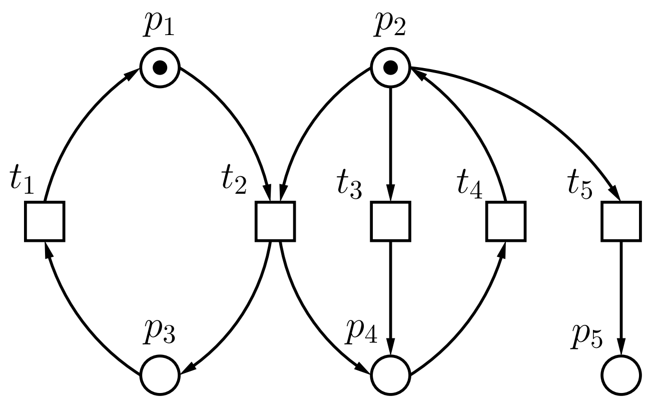

Figure 1: A running example, net system Σ.

A marking M of a net is *n-safe* if $M(p) \leq n$ for every place p. A net system Σ is n-safe if all its reachable markings are n-safe. In this work we mainly consider net systems which are 1-safe. The only exception to this rule is the publication [**P3**], where we also consider n-safe net systems for a fixed integer $n \geq 1$.

Labelled Nets. Let $\mathcal{L}$ be a finite alphabet. A *labelled net* is a pair $\langle N, l\rangle$ (also represented as 4-tuple $\langle P, T, F, l\rangle$), where N is a net and $l\colon\ P \cup T \to \mathcal{L}$ is a labelling function. Notice that different nodes of the net can carry the same label. We extend l to multisets of $P \cup T$ in the obvious way.

For each label $a \in \mathcal{L}$ we define the relation $\xrightarrow{\ a\ }$ between markings as follows: $M \xrightarrow{\ a\ } M'$ if $M \xrightarrow{\ t\ } M'$ for some transition t such that $l(t) = a$.

The reachability graph of a labelled net system $\langle N, l, M_0\rangle$ is obtained by applying l to the reachability graph of $\langle N, M_0\rangle$. In other words, its nodes are the set

$$\{l(M) \mid M \text{ is a reachable marking}\}$$

and its edges are the set

$$\{l(M_1) \xrightarrow{\ l(t)\ } l(M_2) \mid M_1 \text{ is reachable and } M_1 \xrightarrow{\ t\ } M_2\}\ .$$

It should be noted that in this dissertation summary only labelled net systems of a very restricted form are used. Namely, for any two reachable marking M_1, M_2 of the underlying net system $\langle N, M_0\rangle$ such that $l(M_1) = l(M_2)$ it holds that if $M_1 \xrightarrow{\ t\ } M_1'$ then there exist t', M_2' such that $M_2 \xrightarrow{\ t'\ } M_2'$, $l(t') = l(t)$, and $l(M_2') = l(M_1')$.

3 NET UNFOLDINGS

This section introduces net unfoldings and complete finite prefixes more throughly than the presentation in the publications [**P1**]-[**P4**]. The discussion of different notions of prefix completeness in Sect. 3.1 is a subject which is new to the dissertation summary. The presentation of this section is heavily influenced by the presentation of [23, 24, 25].

Net unfoldings were introduced in [58] as a partial order semantics for Petri nets. They were later more extensively researched under the name of branching processes by Engelfriet [16]. McMillan was the first to show how to use net unfoldings as a basis for a verification method [55]. This method has since its publication been the basis of several model checking approaches, some of which we will discuss in this dissertation.

In this section we introduce the definitions needed to describe the unfolding approach. More details can be found in [23, 24, 25, 65, 74].

Occurrence Nets. We use $<_F$ ($\leq_F$) to denote the (reflexive) transitive closure of a flow relation F. We say that two distinct nodes x, y are *causally related*, if $x <_F y$ or $y <_F x$ holds. The nodes x and y are in *conflict*, denoted by $x \# y$, if there exist $t_1, t_2 \in T$ such that $t_1 \neq t_2$, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, $t_1 \leq_F x$, and $t_2 \leq_F y$. The nodes x and y are *concurrent*, denoted by $x \mathit{co}\, y$, if neither $x <_F y$ nor $y <_F x$ nor $x \# y$.

Occurrence nets are nets which have the following special properties. An occurrence net is a net $N = \langle B, E, G \rangle$ such that

- $\forall b \in B : |{}^\bullet b| \leq 1$,
- G is acyclic, or equivalently, $<_G$ is a strict partial order (a transitive and irreflexive relation),
- N is finitely preceded, i.e., for any node x of the net, the set of nodes y such that $y <_G x$ is finite, and
- $\forall x \in B \cup E : \neg(x \# x)$.

The elements of B and E are called *conditions* and *events*, respectively. We also use G instead of F to denote the flow relation of an occurrence net in order to not to cause confusion when the occurrence nets are used later in this section. Let $Min(N)$ denote the set of minimal elements of the strict partial order $<_G$ restricted to the set of conditions. In this work the minimal elements will all be conditions, and thus the set $Min(N)$ can be intuitively seen as an initial marking, called the *default initial marking*. A set of conditions of an occurrence net is a *co-set* iff all the conditions of the set are pairwise is the co relation.

Branching Processes. We associate to a net system Σ a set of *labelled* occurrence nets, called the *branching processes* of Σ. For technical reasons we require that the initial marking M_0 of Σ is 1-safe. The conditions and events of branching processes are labelled with places and transitions of Σ, respectively. The conditions and events of the branching processes are subsets from two sets $\mathcal{B}$ and $\mathcal{E}$, inductively defined as the smallest sets satisfying the following conditions

- $\bot \in \mathcal{E}$, where $\bot$ is an special symbol,
- if $e \in \mathcal{E}$, then $(p, e) \in \mathcal{B}$ for every $p \in P$, and
- if $\emptyset \subset X \subseteq \mathcal{B}$, then $(t, X) \in \mathcal{E}$ for every $t \in T$.

In our definitions of branching process (see below) we make consistent use of these names: The label of a condition (p, e) is p, and its unique input event is e. Conditions $(p, \bot)$ have no input event, i.e., the special symbol $\bot$ is used for the minimal conditions of the occurrence net. Similarly, the label of an event (t, X) is t, and its set of input conditions is X. The advantage of this scheme is that a branching process is completely determined by its sets of conditions and events. This labelling scheme was first introduced by Engelfriet [16].

We will define branching processes inductively in what follows as pairs (B, E) where $B \subseteq \mathcal{B}$ and $E \subseteq \mathcal{E}$. A pair (B, E) can now be alternatively seen as a labelled net system $N = \langle N, l, Min(N)\rangle$ with $N = \langle B, E, G\rangle$ as follows. The labelling l is the one described above, conditions B and events E are as given, and the flow relation G being

- if $e = (t, X) \in E$ and $b \in (X \cap B)$, then $(b, e) \in G$, and
- if $b = (p, e) \in B$ such that $e \in E$, then $(e, b) \in G$.

Note that the definition above does not assume anything about the "consistency" of the labelling, i.e., an edge can only exist if both its endpoints exists. This complicates the definition somewhat but makes it applicable to any pair (B, E) instead of only branching processes. For branching processes the net system as defined above are occurrence net systems, as expected.

We will now inductively define the set of finite branching process of a net system Σ as a pairs (B, E).

Definition 1 *The set of* finite branching processes *of a net system* Σ *with the (1-safe) initial marking* $M_0 = \{p_1, \ldots, p_n\}$ *is inductively defined as follows:*

- $(\{(p_1, \bot), \ldots, (p_n, \bot)\}, \emptyset)$ *is a branching process of* Σ.
- *If* (B, E) *is a branching process of* Σ, $t \in T$, *and* $X \subseteq B$ *is a co-set labelled by* ${}^\bullet t$, *then* $(\, B \cup \{(p, e) \mid p \in t^\bullet\}\ ,\ E \cup \{e\}\,)$ *is also a branching process of* Σ, *where* $e = (t, X)$. *If* $e \notin E$, *then* e *is called a* possible extension *of* (B, E).

The set of branching processes of Σ is obtained by declaring that the union of any finite or infinite set of branching processes is also a branching process, where union of branching processes is defined componentwise on conditions and events. Since branching processes are closed under union, there is a unique maximal branching process, called the *unfolding* of Σ. This result is due to Engelfriet [16]. We will often use the term *prefix* as a synonym for

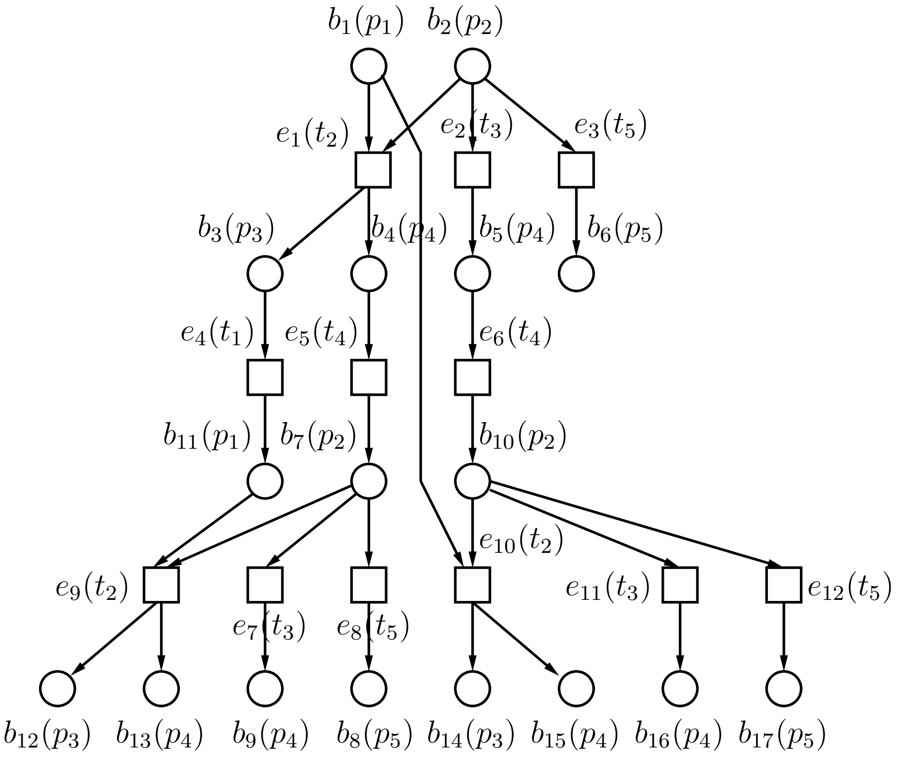

Figure 2: An initial part of the unfolding of Σ.

a branching process of a net system, as any finite branching process can be seen as a prefix of the unfolding of the same net system.

As an example of the unfolding, look at our running example, the 1-safe net system Σ in Figure 1. An initial part of the (infinite) unfolding of Σ is presented in Figure 2. The labelling l is given by the labels in the parentheses.

We take as partial order semantics of Σ its unfolding. This is justified, because it can be easily shown the reachability graphs of Σ and of its unfolding coincide. (Notice that the unfolding of Σ is a *labelled* net system, and so its reachability graph is defined as the *image* under the labelling function of the reachability graph of the *unlabelled* system.) It is possible to show an even stronger correspondence between the behavior of the original net system and the unfolding. Namely, also the partial order behavior of the net system is preserved by unfolding [24, 25].

We often use β to refer to a branching process. Because it is easy to obtain either presentation $\beta = (B, E)$ or $\beta = \langle N, l, Min(N) \rangle$ with $N = \langle B, E, G \rangle$ of the branching process from each other, we use them interchangeably all through this work.

Configurations. A *configuration* of an occurrence net is a set of events C satisfying the two following properties: C is causally closed, i.e., if $e \in C$ and $e' <_G e$ then $e' \in C$, and C is conflict-free, i.e., no two events of C are in conflict. Note that configurations can corresponds to several occurrence sequences of the underlying net system. In our running example in Figure 2 the configuration $C = \{e_1, e_4, e_5\}$ corresponds to the occurrence sequences t_2, t_1, t_4 and t_2, t_4, t_1 of the net system Σ in Figure 1. We call these occurrence sequences *linearisations* of the configuration.

A configuration C of a branching process is associated with a reachable marking of Σ denoted by $Mark(C) = l((Min(N) \cup C^{\bullet}) \setminus {}^{\bullet}C)$. The corresponding set of conditions associated with a configuration is called a *cut*, and is defined as $Cut(C) = ((Min(N) \cup C^{\bullet}) \setminus {}^{\bullet}C)$. Given an event e, we call $[e] = \{e' \in E \mid e' \leq_G e\}$ the *local configuration* of e.

Another way of specifying the unfolding is that it is the output of an unfolding algorithm, Algorithm 1, which was first presented in this form in [24]. We denote by $PE(Unf)$ the set of possible extension $e = (t, X)$ of a branching process $Unf = (B, E)$. For the algorithm to work, we require the following notion of fairness: Whenever an event is added to the set pe of possible extensions, it is also eventually selected to be removed from the set. One possibility to guarantee this is to require that the unfolding algorithm proceeds, e.g., in breadth-first order in generating the unfolding.

Algorithm 1 *The unfolding algorithm*

```
input: A net system $\Sigma = \langle P, T, F, M_0 \rangle$, where $M_0 = \{p_1, \ldots, p_n\}$.
output: The unfolding $Unf = (B, E)$ of $\Sigma$.
begin
$Unf := (\{(p_1, \bot), \ldots, (p_n, \bot)\}, \emptyset)$;
$pe := PE(Unf)$;
while $pe \neq \emptyset$ do
    append to $Unf$ an event $e = (t, X)$ of $pe$ and a condition $(p, e)$
        for every place $p \in t^{\bullet}$;
    $pe := PE(Unf)$;
endwhile
end
```

Note that the unfolding algorithm might not terminate because the unfolding can be an infinite object.

3.1 Complete Finite Prefixes

When we are interested in Petri nets which have only a finite number of reachable states, then the net unfolding will contain a finite initial part, which contains full information about the net unfolding. This observation was first made by McMillan [55], who developed an algorithm to obtain such *complete finite prefix* of the unfolding.

Later Esparza et al. [24, 25] improved McMillan's construction to guarantee that for 1-safe net systems one can obtain a prefix which represents all reachable markings and whose size is bound by the number of reachable states of the original net system. A prefix can in some cases be exponentially smaller than the reachability graph of the system, which makes them interesting for verification purposes.

Marking Completeness. We begin by giving a notion of prefix completeness sufficient to check reachability properties using net unfoldings.

Definition 2 *A branching process β of a net system Σ is* marking complete *if for each reachable marking M of Σ there exists a configuration C of β such that $Mark(C) = M$.*

Esparza et al. have shown that for 1-safe Petri nets it is always possible to create a marking complete prefix which has at most as many events as the net system has reachable markings [24, 25]. (Use for example the prefix generation algorithm of [24, 25] and throw away the so called "cut-off events" as soon as they are encountered.)

Net Structure Completeness. We have defined marking completeness in a way which does not require that each enabled transition of the net system is presented as an event in the prefix. To require this, we define the following (incomparable) notion of completeness.

Definition 3 *A branching process β of a net system Σ is* net structure complete *if for each transition t enabled by some reachable marking M of Σ there exists an event e of β such that $l(e) = t$.*

Usually net structure completeness on its own is not a very interesting property, as it only allows one to recreate all the transitions (which are enabled by some reachable marking) of the net system Σ from the prefix β. However, if one creates a prefix which is marking complete, and then adds one event for each enabled transition not existing in the prefix so far, it is possible to obtain a marking and net structure complete prefix whose number of events is bound by the sum of reachable markings and the number of transitions of the net system Σ.

Completeness. Esparza et.al were the first to define a semantic prefix completeness criterion. We will thus simply refer to it as completeness [24, 25].

Definition 4 *A branching process β of a net system Σ is* complete *if for each reachable marking M of Σ there exists a configuration C of β such that:*

- *$Mark(C) = M$, and*
- *for every transition t enabled in M there exists a configuration $C \cup \{e\}$ such that $e \notin C$ and $l(e) = t$.*

Clearly the unfolding of a net system is always complete. If a finite prefix of the unfolding is complete we call it a *complete finite prefix*. Intuitively if a prefix is complete then the unfolding can be "easily" reconstructed from it. (Of course the unfolding could be also obtained from a net structure complete prefix by e.g., recreating the original net system and then unfolding it. However, we do not consider such and indirect way of constructing an unfolding from a prefix "easy".) A complete finite prefix also contains all the information about the reachability graph of the net system, and thus can be seen as a symbolic representation of the reachability graph. A slightly weaker definition of completeness is presented in [50]. It does also allow the unfolding to be "easily" generated from the prefix if needed.

Strong Completeness. The traditional prefix generation algorithms and some of the model checking algorithms use the notion of a set of *cut-off events* E_{cut}. Recently Vogler et al. [74] have added cut-off events into a notion of prefix completeness in an algorithm independent fashion. Their notion of prefix completeness can be parameterized in several different ways. For simplicity we will fix a set of parameters, and obtain an instance of their notion which can be defined as follows.

Definition 5 *A branching process β of a net system Σ is* strongly complete *if there is a set $E_{cut} \subseteq E$ such that:*

- *for each reachable marking M of Σ there is a configuration $C \subseteq (E \setminus E_{cut})$ of β such that $Mark(C) = M$, and*
- *for each configuration $C' \subseteq (E \setminus E_{cut})$ of β and for each transition t enabled by $Mark(C')$ in Σ, there is an event e of β such that $e \notin C'$, $l(e) = t$, and such that $C' \cup \{e\}$ is a configuration of β (e may be in E_{cut}).*

We call an event e *redundant* if there exists an event $e' \in E_{cut}$ such that $e' <_G e$. It is easy to see that all redundant events can be removed and the branching process still stays strongly complete. We will in the following assume that a strongly complete branching process does not contain any redundant events.

Note that this definition requires each configuration C' without cut-off events to be fully extended by all the transitions enabled by the corresponding marking, not just one representative configuration to be fully extended (compare to Def. 4).

The intuition is that E_{cut} contains a set of events, which do not need to be fired to reach any of the reachable markings of Σ, and thus the prefix can be truncated at any event in the set of cut-off events without losing any reachable markings. However, in a strongly complete prefix this truncation should always manifest itself as a cut-off event left in the prefix. This notion of completeness is strongest of all the notion of completeness discussed in this work. Thus if a prefix is strongly complete it is also complete, marking complete, and net structure complete. Clearly, if we remove all the cut-off events from a strongly complete prefix, we will end up with a marking complete prefix which might no longer be net structure complete.

Because strong completeness criterion implies completeness, it might require more events to be present in the prefix than just ordinary completeness. The drawback of these notions of prefix completeness which require more events to be added to the prefix than just plain marking completeness is that the prefix may need to have also a large number of (usually cut-off) events included to satisfy the additional parts of the used completeness requirement. We do not know of an upper bound on the number of such additional events which would be linearly bounded by the number of markings in the reachability graph.

The McMillan's deadlock checking algorithm [55], Esparza's branching time model checker [17, 32], as well as the deadlock checkers of [48, 56] and [**P1**] rely on having a complete set of cut-off events available and thus require strong completeness. However, the reachability translation of [**P1**] works for any marking complete prefix.

The work of Khomenko and Koutny in [49] presents two different deadlock checking procedures. The first one requires strong completeness, while the the second one works for any marking complete prefix. The intuition behind this translation is to express the deadlock as a reachability of a marking satisfying the formula $dead = \neg \bigvee_{t \in T} \bigwedge_{p \in {}^\bullet t} p$. The latest implementation of the reachability checking procedure described in [**P1**] supports a large variety of formulas to be given as input, including the formula *dead*. The details of the improved implementation have been described in [34]. The main feature of the used translation is that it is linear in the input formula size for all supported formulas. We would like to use marking completeness for all reachability properties in the future, as that enables one to generates smaller prefixes than e.g., strong completeness. We need to experiment with this alternative deadlock checking approach to see whether it is a viable solution when e.g., using the approach of [**P1**].

The complexity results of publication [**P2**] were done for the notion of completeness (Def. 4), however, the prefixes used in the proofs are also strongly complete (Def. 5) and thus the results also hold for this stronger notion of prefix completeness.

We believe that also other, more application specific, notions of prefix completeness will be useful in the design of efficient model checking algorithms with prefixes. Instances of this can be found in the publications [**P3**] and [**P4**].

Prefix generation. We will now describe one algorithm which constructs a complete finite prefix of the unfolding of a bounded Petri net. Actually the generated prefix will also be complete with respect to the strong completeness criterion of Def. 5, see [74]. In this form the algorithm was first presented in [24]. First we will introduce some additional notation.

Given a configuration C, we denote by $\Uparrow C$ the set of events of the unfolding given by $\{e \mid e \notin C \ \wedge \ \forall e' \in C : \neg(e \# e')\}$. Intuitively, $\Uparrow C$ corresponds to the behavior of Σ from the marking reached after executing the events in C. We call $\Uparrow C$ the *continuation* after C of the unfolding of Σ. If C_1 and C_2 are two finite configurations leading to the same marking, i.e., $Mark(C_1) = M = Mark(C_2)$, then $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic, i.e., there is a bijection between them which preserves the labelling of events and the causal, conflict, and concurrency relations (see [24, 25]). The basic idea of the unfolding algorithm is to avoid the construction of such redundant isomorphic copies of the same behavior by truncating the unfolding when possible.

Adequate orders. To implement a complete finite prefix generation algorithm we use the notion of *adequate order* on configurations [24, 25]. Given a configuration C of the unfolding of Σ, we denote by $C \oplus E$ the set $C \cup E$, under the condition that $C \cup E$ is a configuration satisfying $C \cap E = \emptyset$. We say that $C \oplus E$ is an *extension* of C. Now, let C_1 and C_2 be two finite configurations leading to the same marking. Then $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic, as was noted above. This isomorphism, say f, induces a mapping from the extensions of C_1 onto the extensions of C_2; the image of $C_1 \oplus E$ under this mapping is $C_2 \oplus f(E)$.

Definition 6 *A strict partial order $\prec$ on finite configurations of the unfolding of a net system is an* adequate order *if:*

- *$\prec$ is well-founded,*
- *$C_1 \subset C_2$ implies $C_1 \prec C_2$, and*
- *$\prec$ is preserved by finite extensions; if $C_1 \prec C_2$ and $\mathit{Mark}(C_1) = \mathit{Mark}(C_2)$, then the isomorphism f from above satisfies $C_1 \oplus E \prec C_2 \oplus f(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

Note the requirement that $\prec$ is a strict partial order (a transitive and irreflexive relation), as this part is sometimes overlooked in the definition.

Total adequate orders have been presented for 1-safe Petri nets in [24, 25] and for synchronous products of transition systems in [23]. The approach has also been adapted for process algebras in [52], where the authors also define an adequate order. The exact definitions adequate orders for 1-safe Petri nets are somewhat involved and left out of this work, the details can be found in [24, 25, 65].

Unlike the semantic definition of strong completeness, Def. 5, the presented algorithm uses an algorithmic way of computing a set of cut-off events. They are identified as follows.

Definition 7 *An event e of a prefix of the unfolding is a cut-off event if the already constructed part of the prefix contains an event e', such that $\mathit{Mark}([e']) = \mathit{Mark}([e])$ and $[e'] \prec [e]$.*

We call the configuration $[e']$ the corresponding configuration. The intuition behind cut-off events is the following. Because $\mathit{Mark}([e']) = \mathit{Mark}([e])$ we know that their continuations $\uparrow[e']$ and $\uparrow[e]$ are isomorphic. Because $[e']$ was already added to the prefix, we don't have to duplicate the same (isomorphic) behavior after $[e]$ but can safely truncate that branch of the prefix without loosing any information. For the correctness proof of the algorithm, see [24, 25].

It is also possible to use non-local corresponding configurations when finding cut-off events of a prefix. This more refined cut-off criterion was first proposed in [35]. Non-local cut-off criteria have not been implemented in currently available prefix generation tools due to high costs of computing them in an implementation.

A Complete Prefix Generation Algorithm. We can now preset the complete prefix generation algorithm of Esparza et al., Algorithm 2. It is an improved version of the first complete prefix generation algorithm developed by McMillan [55].

The algorithm works as follows. First possible extensions are calculated. Then the algorithm adds events to the prefix in increasing adequate order of their local configurations. It adds to the prefix all events which do not have a cut-off in their local configuration, updating the sets of of possible extensions

Algorithm 2 *A (strongly) complete finite prefix algorithm*

```
input: A $n$-safe net system $\Sigma = \langle P, T, F, M_0 \rangle$, where $M_0 = \{p_1, \ldots, p_n\}$.
output: A complete finite prefix $Fin = (B, E)$ of $Unf$.
begin
$Fin := (\{(p_1, \perp), \ldots, (p_n, \perp)\}, \emptyset)$;
$pe := PE(Fin)$;
$cut\_off := \emptyset$;
while $pe \neq \emptyset$ do
    choose an event $e = (t, X)$ in $pe$ such that $[e]$ is minimal
    with respect to $\prec$;
    if $[e] \cap cut\_off = \emptyset$ then
        append to $Fin$ the event $e$ and a condition $(p, e)$
            for every place $p \in t^\bullet$;
        $pe := PE(Fin)$;
        if $e$ is a cut-off event for $Fin$ then $cut\_off := cut\_off \cup \{e\}$;
    else
        $pe := pe \setminus \{e\}$;
    endif
endwhile
end
```

and cut-off events accordingly. For 1-safe nets the number of non-cut-off event generated by the algorithm is bounded by the number of reachable markings when using the adequate order of [24].

The (strongly) complete prefix generated by the algorithm for our running example is presented in Figure 3. In the figures the cut-off events are marked with crosses. We can obtain a smaller marking complete prefix by just removing all the cut-off events from the prefix of Figure 3.

We refer the reader interested in prefix generation algorithms to [23, 24, 25, 50, 65]. The latest development in them is the use of parallel algorithms [40, 74] for prefix generation.

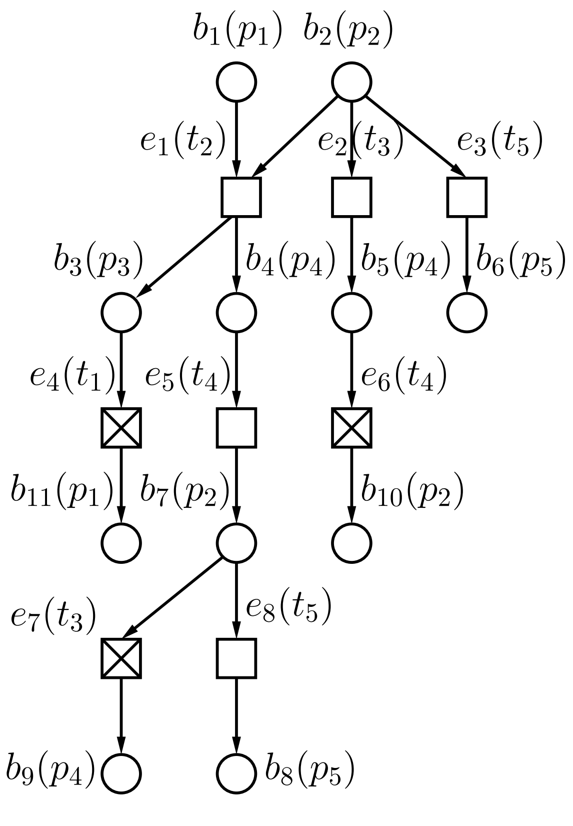

Figure 3: A strongly complete finite prefix of Σ.

4 RULE-BASED CONSTRAINT PROGRAMMING

This section introduces logic programs with stable model semantics in more detail than the publications [**P1**] and [**P6**]. The presentation of this section is based on [67], [**P1**] and slightly extended here. The motivation for using stable models in Section 4.2 is new to this dissertation summary.

We will use normal logic programs with stable model semantics [28] as the underlying formalism into which several verification problems in this work are translated. To be more explicit, the tools developed in publications [**P1**], [**P4**], and [**P6**] employ logic programs with stable model semantics. For a longer introduction to the stable model semantics, and its relation to propositional satisfiability, we refer the interested reader to [59].

The stable model semantics is one of the main declarative semantics for normal logic programs. However, here we use logic programming in a way that is different from the typical PROLOG style paradigm, which is based on the idea of evaluating a given query. Instead, we employ logic programs as a *constraint programming framework* [59], where stable models are the solutions of the program rules seen as constraints.

We consider normal logic programs that consist a set of of rules of the form

$$\mathtt{h} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}, \mathit{not}\,(\mathtt{b}_1), \ldots, \mathit{not}\,(\mathtt{b}_\mathtt{m}) \qquad (1)$$

where $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}, \mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ and $\mathtt{h}$ are propositional atoms. Such a rule can be seen as a constraint saying that if atoms $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}$ are in a model and atoms $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ are not in a model, then the atom $\mathtt{h}$ is in a model. The atom $\mathtt{h}$ is called the *head* of the rule, while the atoms $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}$ and the *not-atoms* $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ are jointly called the *body* of the rule.

The stable model semantics also enforces minimality and groundedness of models. This makes many combinatorial problems easily and succinctly describable using logic programming with stable model semantics.

The stable model semantics for a normal logic program P is defined as follows [28]. (See also an alternative definition after examples below.)

Definition 8 *The reduct P^A of P with respect to a set of atoms A is obtained by*

(i) deleting each rule in P that has a not-atom $\mathit{not}\,(\mathtt{x})$ in its body such that $\mathtt{x} \in A$, and

(ii) by deleting all not-atoms in the remaining rules.

The deductive closure *of P^A is the smallest set of atoms that is closed under P^A when the rules in P^A are seen as inference rules. A set of atoms A is a stable model of P iff A is the deductive closure of P^A when the rules in P^A are seen as inference rules.*

The problem of deciding whether a program has a stable model is NP-complete [54]. The definition above gives a way non-deterministic way of constructing stable models.

We will demonstrate the basic behavior of the semantics using programs P1-P4 below:

$$\begin{array}{llll} \text{P1: } \mathtt{a} \leftarrow \mathit{not}\,(\mathtt{b}) & \text{P2: } \mathtt{a} \leftarrow \mathtt{a} & \text{P3: } \mathtt{a} \leftarrow \mathit{not}\,(\mathtt{a}) & \text{P4: } \mathtt{a} \leftarrow \mathtt{c}, \mathit{not}\,(\mathtt{b}) \\ \quad\ \ \mathtt{b} \leftarrow \mathit{not}\,(\mathtt{a}) & & & \quad\ \ \mathtt{b} \leftarrow \mathit{not}\,(\mathtt{a}) \\ & & & \quad\ \ \mathtt{d} \leftarrow \mathtt{b} \end{array}$$

Program P1 has two stable models: $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$. For example if we guess $A = \{\mathtt{a}\}$ we obtain the reduct program $P^{\{\mathtt{a}\}} = \{\mathtt{a} \leftarrow\}$, whose deductive closure is $\{\mathtt{a}\}$ and thus A is a stable model. The $\{\mathtt{b}\}$ case is symmetric. If we guess $A = \{\mathtt{a}, \mathtt{b}\}$ we obtain the reduct program $P^{\{\mathtt{a},\mathtt{b}\}} = \emptyset$, whose deductive closure is $\emptyset$ and thus A is not a stable model. If we guess $A = \emptyset$ we obtain the reduct program $P^{\emptyset} = \{\mathtt{a} \leftarrow, \mathtt{b} \leftarrow\}$, whose deductive closure is $\{\mathtt{a}, \mathtt{b}\}$ and thus A is not a stable model.

Program P2 has the empty set as its unique stable model. This exposes the fact that the deductive closure of $P^{\emptyset} = \{\mathtt{a} \leftarrow \mathtt{a}\}$ is the empty set. Also the deductive closure of $P^{\{\mathtt{a}\}} = \{\mathtt{a} \leftarrow \mathtt{a}\}$ is the empty set, and thus $A = \{\mathtt{a}\}$ is not a stable model.

Program P3 is an example of a program which has no stable models. If we guess $A = \emptyset$, then we will deduce $\{\mathtt{a}\}$, which will contradict with our assumption $A = \emptyset$, and symmetrically for $A = \{\mathtt{a}\}$.

Program P4 has one stable model $\{\mathtt{b}, \mathtt{d}\}$. If we guess $A = \{\mathtt{b}, \mathtt{d}\}$ we will get the reduct program $P^{\{\mathtt{b},\mathtt{d}\}} = \{\mathtt{b} \leftarrow, \mathtt{d} \leftarrow \mathtt{b}\}$ whose deductive closure is $\{\mathtt{b}, \mathtt{d}\}$. If we guess $A = \{\mathtt{a}, \mathtt{c}\}$ we will get the reduct $P^{\{\mathtt{a},\mathtt{c}\}} = \{\mathtt{a} \leftarrow \mathtt{c}, \mathtt{d} \leftarrow \mathtt{b}\}$ whose deductive closure is $\emptyset$ which does not agree with our guess. Other cases are similar.

Next we proceed to give an alternative definition of the stable model semantics using slightly different notation.

Definition 9 *Let A be a set of atoms, we define $\mathit{not}\,(A) = \{\mathit{not}\,(\mathtt{a}) \mid \mathtt{a} \in A\}$.*

For a set of atoms and not-atoms B we denote the atoms in B by B^+ and the set of not-atoms by B^-. Atoms and not-atoms are also called *literals*. We denote with $\mathit{Atoms}(P)$ the set of all propositional atoms which appear in the logic program P as literals. We use the notation $\overline{\Delta}$ to denote the set $\mathit{Atoms}(P) \setminus \Delta$.

Definition 10 *The deductive closure of a set of rules P and a set of literals B is denoted by $\mathit{Dcl}\,(P, B)$, where $\mathit{Dcl}\,(P, B)$ is the smallest set of atoms that contains B^+ and is closed under $R\,(P, B)$ when*

$$\begin{aligned} R\,(P, B) = \{ & \mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n} \mid \\ & \quad \mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathit{not}\,(\mathtt{b_1}), \ldots, \mathit{not}\,(\mathtt{b_m}) \in P \text{ and} \\ & \quad \mathit{not}\,(\mathtt{b_i}) \in B^-, \text{ for } i = 1, \ldots, m\} \end{aligned}$$

is seen as a set of inference rules.[2]

The deductive closure gives us a fixpoint characterization of the stable models.

[2]This could alternatively be defined as computing the least fixpoint of a suitably defined monotone predicate transformer over the atoms of the program, see e.g., page 6 of [68].

Proposition 1 *The set of atoms Δ is a stable model of a set of rules P iff $\Delta = Dcl\,(P, not\,(\overline{\Delta}))$.*

The proof is immediate by noting that the reduct $P^{\Delta} = R\,(P, not\,(\overline{\Delta}))$.

We explain the notation of the definition above with a non-deterministic algorithm for computing stable models which uses this notation. First one guesses a set of atoms Δ, then computes all atoms of the program not in Δ, i.e., $\overline{\Delta}$. Then all these atoms not in Δ are used as "negative assumptions"[3] $B = not\,(\overline{\Delta})$ to compute the reduct program $P^{\Delta} = R\,(P, B)$. This reduct program does not contain any not-atoms, and it is thus easy to compute the deductive closure Δ' in polynomial time. If the result is the same as our guess, i.e., $\Delta = \Delta'$ we have found a stable model.

There is another way of looking at reducts and deductive closures. Note that the reduct program only contains rules without not-atoms. Therefore the program can be seen as a conjunction of propositional Horn clauses [59]. For example in $P4$ above one of the reduct programs was $P^{\{\mathtt{b},\mathtt{d}\}} = \{\mathtt{b} \leftarrow, \mathtt{d} \leftarrow \mathtt{b}\}$ which can be seen as the propositional formula $((\mathtt{b}) \wedge (\mathtt{d} \vee \neg\mathtt{b}))$. Now the deductive closure of the program is the (unique) subset minimal model of this propositional formula, in this case $\{\mathtt{b}, \mathtt{d}\}$. As there are linear time algorithms to obtain subset minimal models for Horn clauses, their implementation techniques can be applied to computing deductive closures.

Logic programs with stable model semantics has been used to encode several NP-complete problems including combinatorial graph problems such as Hamiltonian circuits and 3-coloring, propositional satisfiability (both conjunctive normal form *CNF* and non-CNF expressions), product configuration, AI planning, and computer aided verification problems as presented in this work, for references see e.g., [59, 61, 68].

4.1 The Smodels System

There is a tool, the Smodels system [60, 68], which provides an implementation of logic programs as a rule-based constraint programming framework. It finds stable models of a logic program, and can also tell when the program has no stable models.

The implementation is based on backtracking search technique similar to the Davis Putnam method (see e.g., [27]), and it uses a generalization of the well-founded semantics [71] to approximate the stable models and to prune the search space. The Smodels implementation needs space linear in the size of the input program [68]. The Smodels seems to be the most efficient implementation of the stable model semantics currently available and it has been applied successfully in a number of areas, for references see e.g., [68].

The stable model semantics is defined using rules of the form (1) above. We employ some extensions, called *extended rules*, which can be seen as compact shorthands for a set of basic rules. The Smodels version 2 handles these extended rules directly [61, 68]. We will now discuss these extensions and their semantics. For an alternative discussion, see Section 3 of [**P6**].

First of the extended rules are *conflict rules*, which are rules of the form: $\mathtt{h} \leftarrow 2\{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$. The semantics of this rule is that if two or more atoms

[3]Note that in our use here $B^{+} = \emptyset$.

from the set $\mathtt{a}_1, \ldots, \mathtt{a}_n$ belong to the model, then also the atom $\mathtt{h}$ will be in the model. It is easy to see that this rule can be encoded by using $\frac{N^2-N}{2}$ basic rules of the form: $\mathtt{h} \leftarrow \mathtt{a}_i, \mathtt{a}_j$.

There is also a linear translation for conflict rules based on adding $\mathcal{O}(N)$ (short, constant size) new rules *and* $\mathcal{O}(N)$ new atoms to the program. The intuition behind this translation is that it is possible to implement a ripple-carry adder style Boolean circuit which outputs true iff less than two of the inputs are true, and the added rules simulate the behavior of this circuit. For a more general way of handling these constraints see Sect. 2.2 of [68].

The conflict rules are very useful for encoding conflict relations between transitions of a Petri net. Having them directly supported by Smodels improved the performance in many cases. This was observed during preliminary experimental work for the publication [**P1**]. When conflict rules were replaced by the $\frac{N^2-N}{2}$ basic rules as described above, Smodels running times were significantly increased.

We also use the so called *integrity rules* in the programs. They are rules with no head, i.e., of the form: $\leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathit{not}(\mathtt{b}_1), \ldots, \mathit{not}(\mathtt{b}_m)$. The semantics is given by the following[4]: First we add two new atoms to the program, call them `contradiction` and `bad`. Next we add new rule to the program: $\mathtt{contradiction} \leftarrow \mathtt{bad}, \mathit{not}(\mathtt{contradiction})$. It is easy to see that any model containing `bad` is not a stable model when this rule has been added. Now we can replace each integrity rule with a rule having `bad` as the head, i.e., the rule becomes: $\mathtt{bad} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathit{not}(\mathtt{b}_1), \ldots, \mathit{not}(\mathtt{b}_m)$. It is easy to see that any set of atoms, such that $\mathtt{a}_1, \ldots, \mathtt{a}_n$ are in a model and atoms $\mathtt{b}_1, \ldots, \mathtt{b}_m$ are not in a model, is not a stable model. It is also easy to see that adding one integrity rule to a program does not create any new stable models, and neither does adding any set of integrity rules.

The last extended rule we use is called a choice rule and is of the following form: $\{\mathtt{h}\} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathit{not}(\mathtt{b}_1), \ldots, \mathit{not}(\mathtt{b}_m)$. The semantics is the following: A new atom $\mathtt{h}'$ is introduced to the program, and the rule is replaced by two rules: $\mathtt{h} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathit{not}(\mathtt{b}_1), \ldots, \mathit{not}(\mathtt{b}_m), \mathit{not}(\mathtt{h}')$, and $\mathtt{h}' \leftarrow \mathit{not}(\mathtt{h})$. The atom $\mathtt{h}'$ is removed from any stable models it appears in, and the rest of the model gives the semantics for the extended rule.

4.2 Motivation for Using Stable Models

In this work logic programs with stable models are used as a formalism into which several NP-complete problems have been mapped into. Here we briefly discuss our motivation for using stable models over other formalisms for solving NP-complete problems like e.g., propositional satisfiability (SAT) or mixed integer programming (MIP).

Our motivations for using logic programs with stable model semantics could be summed up as follows (in no particular order). Some of them are scientific, while others are more of a social nature.

- We needed to efficiently encode conflicts using logic program rules of the form $\leftarrow 2\{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ and a similar construct was not supported by academic SAT systems without a substantial blow-up in formula size

[4]For a 3-SAT translation using this technique, see page 6 of [68].

at the time this work began [33]. Also at that time to our knowledge there were no academic satisfiability checkers which would handle non-CNF formulas.[5] The Smodels could handle both these features nicely. To our knowledge the first academic SAT checker supporting both these features is BCSat [46] which was not yet available when the first publications were created. We also did not have access to the commercial Prover satisfiability checker [66], which has these features.

- While the MIP approach could handle the constraints of the form $\leftarrow 2\{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ we could not find an academic MIP solver with performance approaching that of Smodels. The special purpose MIP solver of [48, 49] does do much better than general purpose solvers but was not available when our work began. Moreover, their algorithms are specific to complete finite prefixes which limits their applicability to other domains.

- The Smodels system has linear memory requirements in the input program size, and combined with our linear size translations of different problems in [**P1**], [**P4**], and [**P6**], we got methods to solve these problems in linear space.

- In the publication [**P6**] the LTL-X translation uses logic programs with stable model semantics in a way which is difficult to translate automatically in a succinct way to SAT.[6] For example, the translation for until formulas contains cyclic dependencies. The fact that the deductive closure of a reduct program is the least fixpoint of a (suitably defined) monotone predicate transformer perfectly matches the fact that the truth value of an until formula can also be defined as a least fixpoint of a monotone predicate transformer. (See also Appendix A, additions to publication [**P6**].) With propositional logic it is easy to express any fixpoint of a monotone predicate transformer, but ensuring that the fixpoint is the least fixpoint is more involved.

- The author was familiar with the stable model semantics and Smodels system already before work on [33] (finally leading to publication [**P1**]) began. Also substantial local knowledge and help was available when using this formalism and the Smodels system.

- The Smodels system was freely available as a C++ library under GPL license, which made it easy to integrate to other tools.

The experience gained from working with Smodels system have resulted in a constrained Boolean circuit satisfiability system BCSat first presented in [46].

With the notable exception of the LTL-X translation part of publication [**P6**] all the logic program translations we have presented can be converted

[5]While not absolutely necessary, we find it easier to use non-CNF satisfiability procedure instead of a CNF one. Of course it is straightforward to translate a constrained Boolean circuits to CNF, but it increases the implementation effort.

[6]Best automatic translations are at least quadratic from logic programs with stable model semantics to SAT in the general case [3].

into constrained Boolean circuits of the same size by simple syntactic translation (and thus also to CNF formulas if need be), for more discussion see Section 3.3 of publication [**P6**]. We believe that such a propositional translation will have slightly better performance due to lower overhead in the solvers used. Starting from scratch we would probably take this route and use, e.g., constrained Boolean circuits with a SAT solver supporting the gate selection used by publication [**P5**].

5 VERIFICATION WITH PREFIXES

This section is a collection of observations about model checking algorithms using complete finite prefixes, and their relation to the research done in the publications [**P1**]-[**P4**]. Most of the presentation in this section is new to the dissertation summary.

Complete finite prefixes can be seen as a symbolic representation of the reachability graph which can sometimes be exponentially more succinct that the reachability graph. This makes it interesting to develop model checking procedures based on complete finite prefixes.

Complete finite prefixes have been used in several different verification tools. McMillan was the first to introduce a deadlock checking procedure using prefixes [55] in a form of a branch-and-bound algorithm. Esparza introduced an branching time logic model checker [17]. An erratum was found in this procedure, which was fixed by Graves [32].

Melzer and Römer in [56] introduced mixed-integer-programming (MIP) as a solution method for deadlock detection using prefixes. We adapt their deadlock checking procedure in publication [**P1**] to logic programs with stable model semantics, and also show how to do reachability checking using the same logic programming methodology. Experimental results can be found in publication [**P1**], and they are quite competitive to both the MIP approach of [56] as well as an implementation of McMillan's deadlock checker described in [56]. Later in experiments of [34] we found out that by enabling the "no-lookahead" option of the underlying logic programming system we could often obtain further (sometimes quite substantial) improvements in deadlock and reachability checking running times. This new improved set of options were also used in the comparisons published in [26, 49]. All the experiments in these publications were obtained by invoking the mcsmodels tool using the option "-n" which enables the "no-lookahead" option in underlying Smodels solver.

In [48] another mixed integer programming approach was given for deadlock checking. It obtains better performance than the authors of [56] by using an application specific search procedure to solve the generated MIP instances. This procedure has been extended to handle also some reachability properties, and in [49] the authors compare against the latest implementation of our deadlock procedure described in [**P1**], [34]. They conclude that even though the two procedures are based on different principles, the performance of the tools are comparable on deadlock checking examples.

In [26] a reachability checker based on explicitly using the co-relation is introduced, as well as a method for checking (some) reachability properties on-the-fly during prefix generation. (The co-relation has been obtained from a prefix generator described in [23].) This work also contains comparisons to the latest implementation of the reachability checking method described in [**P1**], [34]. The authors conclude that when the marking to be checked is reachable the on-the-fly procedure is competitive in several cases, but to show non-reachability our procedure is to be preferred.

The first person to consider model checking linear time temporal logics was Wallner [75]. The logics he used are the action and state based versions of the logic LTL-X, the linear temporal logic without the next time operator.

Unfortunately the version of the procedure published [75] contained an error which was corrected, but the corrected version of the procedure has not so far been published.

In publication [**P3**] we consider model checking an action based version of LTL-X. The procedure is presented in a from of a tableau system which can be alternatively seen as a set of prefixes. In the publication [**P4**] we use a state based LTL-X logic and make the tableau generation procedure more similar to the basic prefix generation algorithm, thus solving several implementation related problems.

5.1 Computational Complexity Issues

In this section we assume the reader to be familiar with basic notions of complexity theory including the complexity classes NP-complete and PSPACE-complete. We use the same terminology and definitions as Papadimitriou in [62], unless explicitly otherwise stated.

It is well known that most verification problems for 1-safe Petri nets such as: reachability of a marking, existence of a reachable deadlock, LTL model checking, and CTL model checking are PSPACE-complete in the size of the net system, for an introduction see e.g., [18].

Reachability with Prefixes. McMillan showed that deadlock checking using a finite complete prefix as input is NP-complete in the size of the prefix [55]. However, the prefix can sometimes be exponentially larger than the 1-safe Petri net from which it was created, thus explaining the difference between the complexity of these two problems. (Using the plausible assumption that NP-complete problems are easier than PSPACE-complete problems.)

Using variations of McMillan's proof one can also show that the reachability problem using complete prefixes as input is also NP-complete in the prefix size [26, 34].

Model Checking with Prefixes. Somewhat surprisingly to us, we were able to show also a negative result for using complete prefixes in model checking. In publication [**P2**] we prove that when the only thing which can be assumed from a prefix is that it fulfills the (strong) completeness criterion, model checking fixed size formulas of several temporal logics is PSPACE-complete in the size of the complete finite prefix. In all of these temporal logics one can express a simple form of nested reachability, a violation of a certain safety property [**P2**]. The proof employs a class of 1-safe Petri nets where a (strongly) complete prefix is only polynomially larger than the original net system, and also easily computable in polynomial time. Thus, intuitively, the complete prefix is sometimes as compact as the original net system. This intuitively makes it "too compact" symbolic representation of the reachability graph for model checking properties involving nested reachability, as we could in principle just use the original net system instead.

To make the checking of nested reachability less complex in the prefix size one can do at least two things which make the prefix larger, but allows one to use less sophisticated algorithms on the obtained (larger) prefix. (i) For linear time properties it is often possible to use techniques from automata theoretic LTL model checking, see e.g., Sect. 9 of [11]. The main idea is to generate a prefix of a suitably defined "product net system" which contains places from both the original net system and e.g., an automaton modeling violations of the property to be checked. In the case of e.g., LTL-X safety properties[7] this product net system will have a reachable marking where, say, a certain place is marked iff the original net system violates the property. Just using this idea basically transforms LTL-X safety model checking into prefix generation of a larger "property specific" product net system. (ii) It is also possible to change the definition of prefix completeness and/or the used prefix generation algorithm to something else than strong completeness. In this case the prefix can have some additional properties which which can make e.g., loop detection easier for them.

Wallner's LTL-X model checking procedure [75] does implement the product approach (i) mentioned above but is still left with a problem of finding certain loops in the behavior induced by the generated product prefix. This turned out to be quite a subtle problem, and the procedure published in [75] contained an error which was later fixed by Wallner but so far left unpublished. The main drawback of the fixed procedure is that it can in some cases use an exponential amount of memory compared to the size of the prefix from which it needed to detect the loops from. The prefixes he uses are generated by Algorithm 2 from a product net system [75]. We do not know if they have some special properties, which would make loop detection easier than for those prefixes just fulfilling the (strong) prefix completeness criterion.

Partly motivated by Wallner's work and the complexity results of [**P2**] we worked on an alternative LTL-X model checking procedure. We used the product approach of Wallner (including also an implementation synchronizing a net system and a Büchi automaton together implemented by Wallner, see [**P4**]). However, we decided to overcome the loop detection problem by modifying the prefix generation algorithm (and thus indirectly also the prefix completeness criterion) to be application specific in the LTL-X model checking publications [**P3**], [**P4**]. This resulted in an application specific definition of a "complete" prefix (also called a tableaux in [**P3**], [**P4**]), but with the advantage that we do not need complex and subtle to implement algorithms to be run after prefix generation. Thus our LTL-X model checking procedure does use both the approaches (i) and (ii) above in order to avoid using difficult and subtle to implement algorithms taking the prefix as input. The drawback is that sometimes the prefixes we generate are larger. However, we were able to prove a bound on their size. The main challenge is to allow as much concurrency as possible in the product net system, in order to sometimes obtain exponential space savings when comparing to a reachability graph based approach, as well as to the approach of Wallner.

[7]Any property whose violation can be expressed by a finite state automaton on finite strings can be handled by this approach. However, to obtain any benefits from net unfoldings one should limit to stuttering invariant properties (for definition, see e.g., Section 10.2 of [11]), as for those a simple synchronization construction which preserves concurrency of "transitions invisible to the automaton" is possible. For a similar discussion and a product construction in the full LTL-X case see [**P4**].

Complete Finite Prefix Generation. In the following we will try to give a glimpse of computational complexity issues in complete finite prefix generation. Clearly it would be advantageous if the problem was well analyzed, as more and more work is based on using complete finite prefix generation as an intermediate phase. Unfortunately, the computational complexity of prefix generation is not yet well understood, and mostly remains an open problem.

Esparza et al. [24, 25] give an upper bound on their algorithm running time in terms of algorithm output size (a strongly complete finite prefix). However, optimally we would like to, e.g., give tight bounds for the amount of memory and/or time needed by an algorithm to output a marking complete finite prefix in terms of the 1-safe net system given as input to a prefix generation algorithm. We do not know of any recent work along these lines.

The most expensive subroutine of the prefix generation algorithm of [24, 25] and its derivatives is the subroutine which calculates the set of possible extensions. It can be show that a decision version of this problem is NP-complete [26, 34], but if the maximum preset size of transitions is fixed this decision version admits a polynomial time algorithm [26, 34]. The possible extensions problem is closely related to a certain clique problem, which has a similar characteristic [26].

6 BOUNDED MODEL CHECKING

This section introduces bounded model checking of asynchronous systems, which is the topic of publications [**P5**] and [**P6**]. The first half of the section is an extended version of bounded model checking introduction from the publications [**P5**], [**P6**]. The second half of is new to the dissertation summary. It discusses the relation of the processes to net unfoldings, and the LTL-X model checking translation of [**P6**].

Bounded model checking [6] has been proposed as a verification method for reactive systems. The main idea is to look for bounded counterexamples, i.e., executions of the system which do not satisfy the specified property, which are shorter than some fixed user supplied length n. Bounded model checking procedures often translate this problem into a propositional satisfiability task. Given the transition relation of the reactive system to be model checked, the property, and the bound n, the transition relation and property are "unrolled" n times to obtain a propositional formula which is satisfiable iff there is a counterexample of length n, see e.g., [6]. The implementation ideas are quite similar to those used in SAT-based AI planning [47, 59].

An important feature of bounded model checking procedures is that they often use significantly less memory than conventional model checkers. It seems that the bounded model checking procedures can currently challenge OBDD based symbolic model checking methods on digital hardware designs both in terms of memory and time consumption needed to find counterexamples [7, 8, 13]. The weakness of bounded model checking is that if no counterexample can be found using some bound n, usually the result is inconclusive. This makes bounded model checking more attractive for "bug hunting" and less attractive for proving systems correct.

In some cases it can be proved that e.g., all reachable markings of a system are reachable within some bound n. In this case, bounded model checking can be used to show that there is e.g., no deadlock in the system by proving that there is no deadlock within bound n. For discussion about how to show completeness for reachability and some other model checking problems, see [6].

Most of the bounded model checking work has so far been dealing with synchronous hardware designs. In this work we deal with asynchronous systems, and more specifically 1-safe Petri nets. When given a description of a 1-safe Petri net system, a 1-safe marking M, and a bound n (encoded in binary) as input, checking whether the marking M is reachable within bound n is obviously PSPACE-complete. This is the case, because the number of reachable markings is at most $2^{|P|}$, all reachable markings are within bound $2^{|P|} - 1$, and this bound can be encoded as compactly as the net system itself. If we encode the used bound n in unary encoding, the problem becomes NP-complete. In practice often the above mentioned worst case bounds do not occur, and this makes bounded model checking feasible.

Asynchronous systems have properties which can be exploited to make bounded model checking for them more efficient. One of the main goals of the publications [**P5**] and [**P6**] is to use the concurrency of the underlying 1-safe Petri net in order to reach states using as small bounds as possible.

The publication [**P5**] considers reachability checking with three different semantics: interleaving, step, and process semantics. The formalism into which bounded reachability checking problems are translated is that of constrained Boolean circuits. In the experiments we employ the BCSat Boolean circuit satisfiability checker [46].

In the interleaving semantics only one transition can fire at one "time step", while step and process semantics both allow a set of transitions to be fired concurrently. The set of reachable states is the same in all three semantics. However, by using concurrency some states can be reached with smaller bounds in step and process semantics.

The idea in step semantics is to allow transitions which are concurrent to be fired simultaneously. For example, if we have a set of n transitions which are concurrent, in step semantics we can reach a state in which all of them have been fired with bound 1, instead of needing a bound of n as in the interleaving semantics. This is also the best case, so only polynomial saving in bound size can be achieved. However, even this can be quite significant, as our experimental results indicate. Note that we do not force concurrent transitions to be fired simultaneously (i.e., our steps are not necessarily maximal steps), and any non-empty subset of transitions in a step is also a step. This implies that all the enabled transitions in the interleaving semantics are steps (containing a single transition) in the step semantics.

There is a partial order semantics for Petri nets called processes [4, 5, 29, 31]. One way of defining processes is through the configurations of the unfolding of a net system. Namely, given a configuration C of the unfolding, a process is the labelled subnet of the unfolding, which contains the minimal conditions of the unfolding and all events in C together with their postset conditions.[8] The process is thus an occurrence net in which no two events are in conflict. It intuitively represents one "concurrent execution" of the net system to the final marking of the process, which is equivalent to the marking $Mark(C)$ (for definition see Section 3) of the corresponding configuration C.

As configurations, also processes can correspond to a super exponential number of interleaving semantics linearisations. In the case of n independent transitions example discussed above, there are $n!$ different linearisations. Because all interleaving executions are also steps, the "linearisation blowup" is even worse in the step semantics case. If we are only interested in the final marking reached by the process, we would optimally want to generate one step execution for each process instead of all the possible step executions which are linearisations of the same process. As we have shown in publication [**P5**], this is indeed possible by employing a suitable normal form of step executions.

As discussed above, there is a close connection between net unfoldings and processes. Intuitively net unfoldings can be seen as a branching time view of the partial order behavior of a system, while processes correspond to a linear time view of the partial order behavior of the system. This connection might be useful in designing new verification techniques. For example, the depth of a marking complete finite prefix (the maximum level of any event, for the definition of level see [**P2**]) gives a bound within which all reachable markings are in process (and thus also step) semantics. It is easy to modify

[8]This mapping is in fact a bijection between processes of a net system and configurations of the unfolding of the same net system.

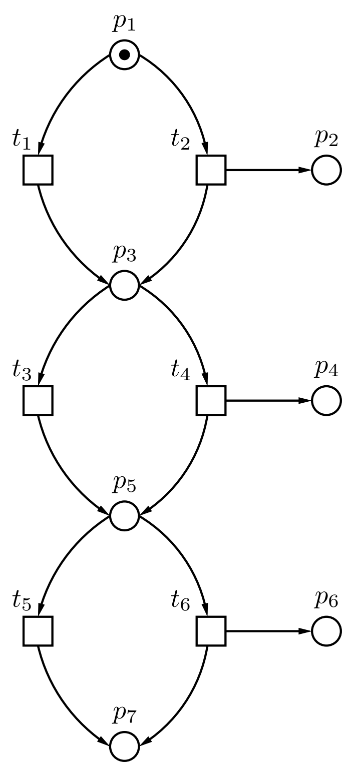

Figure 4: A net system with a large marking complete prefix.

our process semantics translation in such a way that the satisfying models are all processes of depth $0 \leq i \leq n$ instead of processes of exactly depth n as in publication [**P5**]. In this case the process semantics translation of a net system can be seen as a symbolic representation of the all the configurations of the n first levels of the unfolding.

There are sequences of net systems of increasing size n for which the process semantics bounded model checking translation is exponentially more succinct than a marking complete finite prefix of the same net system. For example, Figure 4 contains one instance from such a sequence of net systems for $n = 3$. In this net system all reachable markings are within bound 3 in process semantics. Therefore the process semantics translation grows polynomially in the net system size for net systems in this sequence. However, the marking complete prefix size grows exponentially as the “depth” n of the net systems in this sequence is increased.

The main contribution of the publication [**P6**] is a new more compact LTL-X model checking translation. This translation allows for the concurrency of “transitions invisible to the formula”, thus allowing counterexamples to be sometimes obtained with smaller bounds than with pure interleaving semantics. The SAT based bounded LTL model checking translation presented in [6] is at least quadratic in the formula size, while ours employing logic programs with stable model semantics is linear. The presented LTL-X translation utilizes features of the stable model semantics quite fully as there are, e.g., circular dependencies between atoms in the translation of both *until* and *release* operators. There is no known automatic linear size translation from logic programs with stable model semantics containing such circular dependencies to propositional satisfiability. All known general automatic translations, like the one presented in [3], are at least quadratic. However, the LTL-X translation programs might have some special properties which make a linear size constrained Boolean circuit translation possible. Research along these lines is left for further work.

In the future we would like to combine the good features of the process semantics translation of publication [**P5**] with the LTL-X model checking procedure of [**P6**].

7 CONCLUSIONS

Our research goal has been the development of efficient model checking methods for 1-safe Petri nets. The main problem facing us is the state explosion problem. We have chosen to concentrate on using a combination of symbolic methods and partial order semantics to alleviate this problem. A major part of the work focuses on developing a better understanding of complete finite prefix based verification methods.

In publication [**P1**] a method for solving deadlock and reachability problems of 1-safe Petri nets using a complete finite prefix as input is presented. These problems are translated into the problem of finding a stable model of a logic program, and use the Smodels system [68] as a computational engine. The procedure has been implemented in a tool called mcsmodels.

In the publication [**P2**] several model checking problems using complete finite prefixes as input are shown PSPACE-complete when logics which can express a simple nested reachability property are used.

Publications [**P3**] and [**P4**] contain net unfolding based procedures to model check LTL-X formulas of action and state based versions of the logic, respectively. A prototype of the state based LTL-X model checker has been implemented in a tool called unfsmodels as a variant of a conventional complete finite prefix generation procedure.

Publication [**P5**] gives a bounded model checking translation from 1-safe Petri nets into constrained Boolean circuits. It supports the checking of reachability based properties using process, step, and interleaving semantics. The process semantics version is the main contribution of the publication. The translation has been implemented in a tool called punroll. In the experiments we use the BCSat system [46] to solve the generated constrained Boolean circuit satisfiability problems.

Publication [**P6**] gives a similar bounded model checking translations for step and interleaving semantics but now using logic programs with stable model semantics. The main contribution of the paper is a new, more succinct, LTL-X model checking translation which allows for the concurrency of invisible transitions. The translations have been implemented in a tool called boundsmodels. In the deadlock checking experiments we use the Smodels system [68]. Experimenting with the LTL-X translation is left for further work.

This work contains a selection of model checking approaches for 1-safe Petri nets. The bounded model checking approaches of publications [**P5**], [**P6**] are most interesting in early stages of system development, when one expects a number of counterexamples to be present. They also have the smallest memory requirements of the methods presented in this work. Their weakness is that it is hard to prove systems correct using bounded model checking techniques. After no more errors can be found with bounded model checking techniques, the other presented approaches should be considered. For properties which can be expressed either as a reachability and as an LTL-X properties, the specialized reachability checker of [**P1**] is to be preferred over the full LTL-X model checking approach approach of [**P4**], as the generated prefixes and thus also memory requirements are potentially smaller in this approach.

The strengths of the methods presented in this work are in cases where there is a large amount of concurrency in the system. This concurrency can often be exploited by the presented model checking techniques. The bounded model checking approaches work at their best when the reachable states of a system are within a small bound from the initial state of the system. The performance of the LTL-X model checking using net unfoldings is at its best when the model checked properties are local to some component of the system.

The weaknesses of the proposed methods are in cases when there is little or no concurrency in the system. In the case of complete finite prefix based methods a high degree of nondeterminism resulting in conflicts in the unfolding can result in large prefixes. For an example of net system having both these properties, see Fig. 4 of Section 6. The methods we have presented in this work are for 1-safe Petri nets, which are quite a low level modeling formalism. More work is needed in handling higher level modeling languages which could reduce the modeling effort needed.

Our experimental work has tried to demonstrate the feasibility of the used approaches. More benchmarking is needed, especially against other methods for alleviating the state explosion problem, like partial order reductions and OBDD based methods.

7.1 Topics for Further Research

There are interesting topics for further research.

With the exception of the LTL-X translation part of [**P6**], all our translations can be converted into constrained Boolean circuits of the same size as the logic program version. By doing this one could experiment with a larger class of solvers, and possibly obtain some speed-ups.

On the complete finite prefix side, the computational complexity of algorithms doing prefix generation is not yet well understood. For the safety subset of LTL-X a simple special purpose model checking algorithm using complete finite prefixes could be developed. This can in some cases be more efficient than handling the safety LTL-X properties using a model checker for the full LTL-X logic.

On the bounded model checking side we would like to have a simple succinct LTL-X translation also using constrained Boolean circuits. Combining process semantics with LTL-X model checking is also left for further work. Methods for proving that, e.g., a bound n is sufficient for reaching all reachable markings need to be researched further. A candidate method for doing this is to look at the translation of this problem into a quantified boolean formula (QBF) [6] (for definition, see e.g., QSAT problem in [62]), and at solvers for QBF formulas which arise from this translation.

A CORRECTIONS AND ADDITIONS TO PUBLICATIONS

Publication [P1]:

- On page 249, Section 2.2 Occurrence Nets: "F is a partial order" should be "F is a *strict* partial order".

- On page 250, Section 2.4 Finite Complete Prefixes: For deadlock checking using the translation presented in the paper we should require strong completeness (Def. 5 of dissertation summary Section 3.1) instead of completeness (Def. 4 of dissertation summary Section 3.1) as is done in the publication. All the prefix generation algorithms mentioned in the publication [**P1**], including the one used for experiments, do create strongly complete prefixes. See dissertation summary Section 3.1 for further discussion about different notions of prefix completeness.

- On page 252, Section 3, Definition 3.2, typographical error in definition of $R\,(P, B)$:

  $$\begin{aligned} R\,(P,B) = \{\mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathit{not}\,(\mathtt{b_1}), \ldots, \mathit{not}\,(\mathtt{b_m}) \in P \text{ and} \\ \mathit{not}\,(\mathtt{b_i}) \in B^-,\ \text{for } i = 1, \ldots, m\} \end{aligned}$$

  should be

  $$\begin{aligned} R\,(P,B) = \{\mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n} \mid & \\ & \mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathit{not}\,(\mathtt{b_1}), \ldots, \mathit{not}\,(\mathtt{b_m}) \in P \text{ and} \\ & \mathit{not}\,(\mathtt{b_i}) \in B^-,\ \text{for } i = 1, \ldots, m\} \end{aligned}$$

- On page 256, Section 5 Deadlock Property Checking Implementation: The reachability translation described in the paper was implemented in a later version of the tool, which is called mcsmodels [34]. This version also supports a larger set of reachability properties than just assertions, for implementation details and updated (more competitive) experimental results obtained by using the "no-lookahead" option of the underlying Smodels solver, see [34].

- On page 257, Section 5 Deadlock Property Checking Implementation: Note that some of the described optimizations are deadlock checking specific. For a list of reachability checking optimizations implemented in the mcsmodels tool, see [34].

- On page 260, Section 6 Conclusions, line 9: "one-to-one" should be "bijective". The same error also appears on the last sentence of page 266, and on line 7 of page 267.

- On page 267, line 1, proof of Theorem 4.2:
  "The fact that β is a finite complete prefix of a 1-safe net system Σ guarantees the following. For each reachable marking M of Σ there exists a configuration C of β with no cut-off events, such that $Mark(C) = M$, and for every transition t enabled in M there exists a configuration

$C \cup \{e\}$ such that $e \notin C$ and $h(e) = t$."
should be
"The fact that β is a finite *strongly* complete prefix of a 1-safe net system Σ guarantees the following. For each reachable marking M of Σ there exists a configuration C of β with no cut-off events, such that $Mark(C) = M$, and *for each such configuration C and* for every transition t enabled in M there exists a configuration $C \cup \{e\}$ such that $e \notin C$ and $h(e) = t$. "

Publication [P2]:

- Note that all prefixes used in the proofs of this publication are also strongly complete (Def. 5 of dissertation summary Section 3.1), and thus the complexity results also hold for this notion of prefix completeness.

- On page 115: Note that we use the size of the adjacency matrix representation of the flow relation as the size of the net system.

- On page 120: To prove Lemma 3 it is sufficient to note that the prefix $\beta_C(A)$ consists of the two first levels of the unfolding of $C(A)$ and that all reachable markings are reachable by configurations containing only events from the first level of the unfolding. (Use the same proof as in the example of Section 3, page 113, text below Fig. 3.)

- On page 120, discussion of Theorem 4: To prove that CTL* model checking is in PSPACE for 1-safe Petri nets we simulate the behavior of the Petri net by a concurrent program as defined on page 47 of [51]. (They are basically a synchronization of a set of labelled transition systems.) We can then use the result that model checking a fixed size CTL* formula is PSPACE-complete (and thus in PSPACE) in the size of the description of a concurrent program [51]. Here we give a proof sketch. We first do the following: Add complement places for all places in the net system Σ obtaining only a polynomially larger net system Σ'. We can now decompose this net system in a concurrent program by decomposing the net system Σ' into "state machine components" in the standard way. Each pair of complementary places is turned into a (two-state) state machine. For each place p and each transition t of Σ such that $p \in ({}^{\bullet}t \cup t^{\bullet})$ we add a local transition into the concurrent program which simulates the effect of the transition t on p and its complement place $\overline{p}$, and label this local transition by t. We use the initial state of the Petri net system Σ' as the initial state of created the concurrent program. Now the synchronization of all the local transitions labelled by t is possible iff the transition t is enabled in Σ. Also the combined effect of these local transitions simulates the firing of t. Thus this concurrent program will simulate the behavior of the net system Σ, and we can answer CTL* model checking questions from the reachability graph of this concurrent program. The concurrent program is polynomial in the size of Σ and obtainable by a polynomial time algorithm, thus completing the proof.

- On page 120, discussion of Theorem 4: To prove that the linear time μ-calculus model checking is in PSPACE for 1-safe Petri nets and for a fixed formula, we note that [14] gives a characterization of the logic using Büchi automata. We can then use a Büchi emptiness checking algorithm for 1-safe Petri net systems [18] with this (fixed size) Büchi automaton to obtain a PSPACE algorithm.

Publications [P3] & [20]:

- On page 476 of [**P3**], Section 1 Introduction, and also on page 485 of [**P3**], Section 7 Conclusions: Note that the $\mathcal{O}(K^2)$ upper bound on the number of non-cut-off events of the tableaux holds because the formula is considered to be fixed, and thus contributing only as a constant factor to the tableaux size.
  (Also present in [20].)

- On page 477 of [**P3**], Section 2 Automata theoretic approach to model checking LTL: The set of invisible transitions is $T \setminus V$ used before defined. Add the sentence to the end of line 6: "The set of transitions T is divided into *visible transitions* V and *invisible transitions* $T \setminus V$."

- On page 477 of [**P3**], Section 2 Automata theoretic approach to model checking LTL: Note that we interpret the linear time temporal logic only over infinite sequences. We would like to assume that the net systems given as input to the model checking procedure are deadlock free. If the net system is not deadlock free, our model checker still works correctly in the following sense. We still report counterexamples which are infinite executions. However, we do *not* extend finite executions into infinite ones by adding an infinite loop of dummy transitions at each deadlocked marking as some other linear time temporal logic model checkers do!

- On page 480 of [**P3**], Section 3 Basic definitions on unfoldings: Definition of a continuation is incorrect, replace

  "Given a configuration C, we denote by $\uparrow C$ the set of events e such that (1) $e' < e$ for some event $e' \in C$, and (2) e is not in conflict with any event of C."

  with

  "Given a configuration C, we denote by $\uparrow C$ the set of events of the unfolding $\{e \mid e \notin C \;\wedge\; \forall e' \in C : \neg(e \# e')\}$."

  (Also present in [20].)

- On page 480 of [**P3**], Definition 2:
  "A partial order $\prec$"
  should be
  "A *strict* partial order $\prec$"
  (Also present in [20].)

- On page 482 of [**P3**], Section 4 A tableau system for the illegal ω-trace problem, line 12:
  "doesn't contains"
  should be
  "does not contain"
  (Also present in [20].)

- On page 482 of [**P3**], Section 5 A tableau system for the illegal livelock problem, line 1 of the section:
  To correct the proof of Theorem 9 on pages 29-30 of [20] (see below) we change the definition of checkpoints to a sequence from a set:

  "The tableau system for the illegal livelock problem is a bit more involved that that of the illegal ω-trace problem. In a first step we compute a set $CP = \{M_1, \dots , M_n\}$ of reachable markings of Σ, called the set of *checkpoints*. This set has the following property . . . "

  should be

  "The tableau system for the illegal livelock problem is a bit more involved that that of the illegal ω-trace problem. In a first step we compute a sequence $CP = M_1, \dots , M_n$ of reachable markings of Σ, called the *checkpoints*. This sequence of markings has the following property . . . "

  (Also present in [20].)

- On page 483 of [**P3**]:
  Note that Definition 4 fits the idea of computing checkpoints described in the beginning of the Section 5, as L-transitions are never concurrent with V-transitions.

- On page 483 of [**P3**], Definition 4:
  The proof of Theorem 9 on page 29 of [20] incorrectly assumes that the events $e_1, e_2, \dots , e_m$ are arranged in $e_1 \preceq e_2 \preceq \cdots \preceq e_m$ order in the sentence: "Without loss of generality, we choose (C, e) so that the index i is minimal.". In order not to change the proof substantially, we change the Definition 4:

  "A marking M belongs to the set CP of *checkpoints* of Σ if $M = Mark([e])$ for some non-terminal event e of the complete prefix of Σ labelled by a transition of L."

  to an new version which matches the assumption made by the proof

  "Let $e_1 \preceq e_2 \preceq \cdots \preceq e_m$ be the set of non-terminal events of the complete prefix of Σ labelled by a transition of L ordered in non-decreasing $\preceq$ order. We define the set of checkpoints to be a sequence of markings $CP = M_0, \dots , M_n$, such that:

  - $CP_0 = \epsilon$, and

  - for all $1 \leq i \leq m$: if $Mark([e_i]) \in CP_{i-1}$ then $CP_i = CP_{i-1}$, else $CP_i = CP_{i-1} \cdot Mark([e_i])$.

  Finally we define CP to be CP_m. "

  (Also present in [20].)

- On page 483 of [**P3**], last line of Section 5.1 Computing the set of checkpoints:
  "So $CP = \{ \{p_2, p_4\}, \{p_4, p_7\} \}$."
  should be
  "So $CP = \{p_2, p_4\}, \{p_4, p_7\}$."
  (Also present in [20].)

- On page 483 of [**P3**], first line of Section 5.2 The tableau system:
  "Let $\{M_1, \ldots, M_n\}$ be the set of checkpoints ... "
  should be
  "Let $M_1, \ldots, M_n$ be the sequence of checkpoints ... "
  (Also present in [20].)

- On page 483 of [**P3**], Section 5.2 The tableau system, 3rd line from the bottom of the page. Remove the sentence:
  "The definition of repeat depends on the order of the checkpoints, but the tableau system defined above is sound and complete for any fixed order."
  (Also present in [20].)

- On page 28 of [20], Definition 17:
  Replace
  "An *L-pair* is a pair (C, e) where C is a configuration ... "
  with
  "An *L-pair* is a pair (C, e) where C is a *finite* configuration ... "

- On page 28 of [20], Lemma 18 (1):
  The Lemma 18 (1) is partly incorrect, replace

  "(1) Σ has an illegal livelock $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1}$ if and only if its unfolding contains an L-pair (C, e) such that $Mark([e]) = M$."

  with a weaker but still sufficient claim

  "(1) Σ has an illegal livelock $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1}$ if and only if its unfolding contains an L-pair (C, e).

  Additionally, if the unfolding of Σ contains an L-pair (C, e) then Σ has an illegal livelock such that $M = Mark([e])$."

- On page 28 of [20], Proof of Lemma 18(1)($\Rightarrow$):
  The argument

  "Let C be an (infinite) configuration of the unfolding of Σ such that $\sigma\sigma_1$ is one of its linearisations. C contains an event e corresponding to the last transition of σ. Since C is infinite, $C \setminus [e]$ contains infinitely many events. Let $[e] \oplus E$ be an extension of $[e]$ such that E contains at least $(K \cdot B) + 1$ events. ”

should for consistency be

“Let C be a finite configuration of the unfolding of Σ such that $\sigma\sigma'$ is one of its linearisations, where $\sigma_1 = \sigma'\sigma''$ and σ' contains $(K \cdot B) + 1$ events. C contains an event e corresponding to the last transition of σ. Let $[e] \oplus E$ be an extension of $[e]$ such that $C = [e] \oplus E$.”

- On page 30 of [20], argument on lines 1-14 is incorrect, replace the argument:

“If C_e contains a successful terminal of $\mathcal{T}_i$, then we are done. Otherwise, by Lemma 19, it contains an unsuccessful terminal d. Let d' be the companion of d.

We first prove that d' is also an event of $\mathcal{T}_i$. Assume the contrary. Then, by the definition of a terminal, d' is an event of $\mathcal{T}_j$ and $j < i$. Since d is an event of $\mathcal{T}_i$, we can split σ_1 into two sequences, $\sigma_1 = \sigma_2\sigma_3$, such that σ_2 is a linearisation of $[d]$, and so

$$M_0 \xrightarrow{\sigma} M_i \xrightarrow{\sigma_2} \mathit{Mark}([d]) \xrightarrow{\sigma_3}$$

Since $\mathit{Mark}([d']) = \mathit{Mark}([d])$, we find a linearisation σ_2' of $[d']$ in $\mathcal{T}_j$ such that
$M_j \xrightarrow{\sigma_2'} \mathit{Mark}([d']) = \mathit{Mark}([d])$. So we have

$$M_0 \xrightarrow{\sigma'} M_j \xrightarrow{\sigma_2'} \mathit{Mark}([d']) = \mathit{Mark}([d]) \xrightarrow{\sigma_3}$$

which is an illegal livelock of Σ. Since M_j is a checkpoint and $j < i$, we reach a contradiction to our assumption that the index i is minimal.”

with a more detailed argument:

“If C_e contains a successful terminal of $\mathcal{T}_i$, then we are done. Otherwise, by Lemma 19, it contains an unsuccessful terminal d. Let d' be the companion of d.

We first prove that d' is also an event of $\mathcal{T}_i$. Assume the contrary. Then, by the definition of a terminal, d' is an event of $\mathcal{T}_j$ and $j < i$.

Next we create an illegal livelock execution from events of C_e using exactly the same idea as proof of Lemma 18(1)($\Leftarrow$). As C_e contains at least $(K \cdot B) + 1$ events, by Lemma 14, C_e contains a chain $f_1 < \ldots < f_{K+1}$. By the pigeonhole principle, there are events $f_i < f_j$ such that $\mathit{Mark}(f_i) = \mathit{Mark}(f_j)$. Let $M_i \xrightarrow{\sigma_2} M_1 \xrightarrow{\sigma_3} M_2$ be a linearisation of $[f_j]$ such that $M_i \xrightarrow{\sigma_2} M_1$ is a linearisation of $[f_i]$. We have $M_1 = M_2$, and so $M_i \xrightarrow{\sigma_2} M_1 \xrightarrow{\sigma_3} M_2 = M_1 \xrightarrow{{\sigma_3}^{\omega}}$ is an infinite firing sequence containing only invisible transitions.

Because C_e does not contain a successful terminal of $\mathcal{T}_i$, in particular it does not contain f_j, but contains an unsuccessful terminal $d'' < f_j$.

Now, without loss of generality, we can choose to consider only the case $d = d''$, and thus $d < f_j$.

Now clearly $d \in [f_j]$. Let $M_i \xrightarrow{\sigma_4} M'_1 \xrightarrow{\sigma_5} M_2$ be a linearisation of $[f_j]$, such that $M_i \xrightarrow{\sigma_4} M'_1$ is a linearisation of $[d]$. Thus we get an infinite execution

$$M_i \xrightarrow{\sigma_4} M'_1 \xrightarrow{\sigma_5} M_2 \xrightarrow{{\sigma_3}^{\omega}}$$

Let C'''_e now be a configuration of BP_i which consist of the events of the firing sequence $M_i \xrightarrow{\sigma_4\sigma_5({\sigma_3}^{(K\cdot B)+1})}$. Now clearly $C'''_e \setminus [d]$ contains at least $(K \cdot B) + 1$ events.

Define $C'_e = [d'] \oplus f(C''_e \setminus [d])$, where f is an isomorphism from $\uparrow [d]$ to $\uparrow [d']$.

Define $C' = [e_j] \oplus f_j^{-1}(C'_e)$, where f_j is the isomorphism from $\uparrow [e_j]$ to the unfolding of Σ_j.

Now $(C', [e_j])$ is a minimal L-pair as there are at least $(K \cdot B) + 1$ invisible events in $C' \setminus [e_j]$. Because $j < i$, we reach a contradiction to our assumption that the index i is minimal."

- On page 30 of [20], line 23:
  Replace
  "C'_e is a configuration of $\mathcal{T}_i$."
  with
  "C'_e is a configuration of BP_i."

- On page 30 of [20], line 26:
  Replace
  "Because $\mathcal{T}_i$ is a branching process ... "
  with
  "Because BP_i is a branching process ... "

Publications [P4] & [22]:

- On page 37 of [**P4**], Section 1 Introduction, line 8 in the section:
  "A new algorithm with better complexity bounds was introduced in [19], in the shape of a tableau system."
  should be
  "A new algorithm with smaller memory requirements in several interesting model checking instances was introduced in [19], in the shape of a tableau system."
  (Also present in [22].)

- On page 41 of [**P4**], Theorem 1, footnote:
  Here we shortly describe the problem behind the technical restriction. The technical requirement of incomparable markings comes from the fact that in the algorithm and proofs the L-events should not be concurrent with any other events. In the algorithm on page 50 exactly one L-event (which is not concurrent with any other events) is generated for each local configuration, while there might be several other

L-transitions enabled in $\Sigma_{\neg\varphi}$ for (proper subsets of) the corresponding marking of this local configuration. However, it suffices to generate only the L-event with the maximal marking, as any illegal livelocks found after other L-events will also be found after this generated L-event. We could update the proofs to take this into consideration, but it complicates the theory and proofs somewhat.
(Also present in [22].)

- On page 46 of [**P4**], Definition 2:
  "A partial order $\prec$"
  should be
  "A *strict* partial order $\prec$"
  (Also present in [22].)

- On page 22 of [22], case (d) of proof of Lemma 1:
  Hyphens missing repeatedly due to a technical problem:

  "Since the preset of t is also a reachable marking (q, s_f, O, H), we have $q = q$, $s = s_f$, $O \subseteq O$, and $H \subseteq H$. Since all reachable markings of Σ are pairwise incomparable and (O, H), (O, H) are reachable markings of Σ, we have $O = O$ and $H = H$."

  should be

  "Since the preset of t is also a reachable marking (q', s_f, O', H'), we have $q' = q$, $s = s_f$, $O' \subseteq O$, and $H' \subseteq H$. Since all reachable markings of Σ are pairwise incomparable and (O, H), (O', H') are reachable markings of Σ, we have $O' = O$ and $H' = H$."

- On page 26 of [22], line 5:
  "Notice that σ_2 is a non-empty sequence."
  should be
  "Notice that σ_3 is a non-empty sequence."

- On page 23 of [22], Proof of Theorem 2:
  Notice that the case (1) also covers the case that e is an L-event, as all L-event are always in $BL(C)$.

- On page 24 of [22], Proof of Lemma 2(2):
  Add an argument to the end of the proof: "Because from Lemma 1(e) we get that $[e]$ does not contain any L-events, and thus does not contain any type II terminals."

- On page 26 of [22], line 9:
  "σ_2 contains only invisible actions"
  should be
  "σ_2 and σ_3 contain only invisible actions"
  (The same correction should also be made on the 4th line from the bottom of page 26.)

- On page 27 of [22], Proof of Lemma 6, line 2.
  Typographical error, replace

"$\mathit{Mark}(e') = \mathit{Mark}(e)$"
with
"$\mathit{Mark}([e']) = \mathit{Mark}([e])$".

- On page 27 of [22], Proof of Theorem 7, part (2), lines 3-12.
The argument here is incorrect, replace the argument:

  "By Lemma 3(3), $\Sigma_{\neg\varphi}$ has an illegal livelock $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1}$ such that σ is a linearisation of $BL(C)$.

  We split σ_1 into two sequences, $\sigma_1 = \sigma_2\sigma_3$, such that $\sigma\sigma_2$ is a linearisation of $[d]$, and so

$$M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_2} \mathit{Mark}([d]) \xrightarrow{\sigma_3}$$

  Now find a linearisation $\sigma'\sigma_2'$ of $[d']$ such that
  $M_0 \xrightarrow{\sigma'} M' \xrightarrow{\sigma_2'} \mathit{Mark}([d'])$.
  So we have

$$M_0 \xrightarrow{\sigma'} M' \xrightarrow{\sigma_2'} \mathit{Mark}([d']) = \mathit{Mark}([d]) \xrightarrow{\sigma_3}$$

  which is an illegal livelock of $\Sigma_{\neg\varphi}$.
  By Lemma 5, there is a L-configuration C' such that σ' is a linearisation of $BL(C')$."

  with a new argument:

  "We first use the same procedure as the proof of Lemma 5($\Leftarrow$):

  Let $M_0 \xrightarrow{\sigma} M_1$ be a linearisation of $BL(C)$. By Lemma 3(1), the last transition of σ is labelled by a transition of L. Since e is a successful terminal of type II(b) and C is an L-configuration, there exists an event $e' \in AL(C)$ such that $\mathit{Mark}([e']) = \mathit{Mark}([e])$. Let $M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_2} M_2 \xrightarrow{\sigma_3} M_3$ be a linearisation of $[e]$ such that $M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_2} M_2$ is a linearisation of $[e] \cap [e']$. By Lemma 4 we have $M_2 = M_3$, and therefore we have an illegal livelock
  $M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_2({\sigma_3}^{\omega})}$.

  Because $d < e$ we also have an illegal livelock

$$M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_4} M_2' \xrightarrow{\sigma_5} M_3 \xrightarrow{{\sigma_3}^{\omega}}$$

  where $M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_4} M_2'$ is a linearisation of $[d]$ and the execution $M_0 \xrightarrow{\sigma} M_1 \xrightarrow{\sigma_4} M_2' \xrightarrow{\sigma_5} M_3$ is a linearisation of $[e]$.

  Now find a linearisation of $[d']$

$$M_0 \xrightarrow{\sigma'} M_1' \xrightarrow{\sigma_2'} M_2''$$

  such that $M_0 \xrightarrow{\sigma'} M_1'$ is a linearisation of $BL([d'])$.

  So we have

$$M_0 \xrightarrow{\sigma'} M_1' \xrightarrow{\sigma_2'} \mathit{Mark}([d']) = \mathit{Mark}([d]) \xrightarrow{\sigma_5} M_3 \xrightarrow{{\sigma_3}^{\omega}}$$

which is an infinite execution of $\Sigma_{\neg\varphi}$. Clearly the last transition of σ' belongs to L by Lemma 3(1), and the fact that d' is a part-II event. Thus this execution is also an illegal livelock.

By Lemma 5, there is a L-configuration C' such that σ' is a linearisation of $BL(C')$."

- On page 28 of [22], line 13:
  "Finally, $C' \prec_{LTL} C$ is proved exactly as in case (2)."
  should be
  "Finally, $C' \prec_{LTL} C$ is proved directly from the definition of $\prec_{LTL}$."

- On page 28 of [22], 2nd line after Theorem 8:
  "can be divided into to disjoint sets"
  should be
  "can be divided into *two* disjoint sets"

Publication [P5]:

- Note that the process semantics translation can be seen as a system with a history dependent transition relation (a transition is only enabled if some transition at the previous time step generated a token to its preset). Another way of looking at this is that the places of the net can be in three different states: (i) having no token, (ii) having a token generated in the previous step, or (iii) having a token generated in some earlier step. Now the process semantics translation can be seen as a (suitably defined) transition relation of a system with at most $3^{|P|}$ states (instead of $2^{|P|}$ states as in the case of step semantics). Notice, however, that for any given bound n the executions of the process semantics translation are a (often proper) subset of the step executions. Thus for bounded model checking the system with a larger possible reachability graph actually has less behavior than the system with a smaller possible reachability graph for a given bound n.

- On page 221, Definition 1, 3rd line from the bottom of the page:
  Typographical error, replace "$l_{i+i}(b'(p))$" with "$l_{i+1}(b'(p))$".

Publication [P6]:

- Our LTL-X translation behaves quite differently for counterexamples without a loop, and counterexamples with a loop. In the case of counterexamples without a loop, our translation becomes equivalent to the without loop case of [6].

- In the case of counterexamples with a loop our translation differs significantly from the one presented in [6]. We use the fact that for Kripke structures in which each state has exactly one successor, the semantics of CTL and LTL coincide, see e.g., Theorem 1 of [69]. In the semantics of CTL an existential until can be defined as a least fixpoint of a monotone predicate transformer, see e.g., Section 7.5 of [11]. Now the translation for until formulas uses the fact that the deductive closure of a reduct program is the least fixpoint of a suitably defined monotone predicate transformer, actually in this case the predicate transformer defining until. The translation for release formulas uses the until translation together with a specialized translation for $f = \Box f_1$ formulas, which uses the fact that the duality $\Box f_1 \equiv \neg\Diamond\neg f_1$ holds if a loop exists. (Note that this duality does not hold if a loop does not exists, see [6].)

- Note that a naive SAT version of the LTL-X translation will be incorrect because of cyclic dependencies used by the translation. The translation depends on the least fixpoint interpretation of these cyclic dependencies in the stable model semantics.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proceeding of 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems, (TACAS'2000)*, pages 411–425. Springer-Verlag, 2000. LNCS 1785.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[3] R. Ben-Eliyahu and R. Dechter. Default reasoning using classical logic. *Artificial Intelligence*, 84:113–150, 1996.

[4] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.

[5] E. Best and C. Fernández. *Nonsequential Processes: A Petri Net View*, volume 13 of *EATCS monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer-Verlag, 1999. LNCS 1579.

[7] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In *Proceeding of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 60–71. Springer-Verlag, 1999. LNCS 1663.

[8] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of 13th International Conference on Computer Aided Verification, (CAV'2001)*, pages 454–464. Springer-Verlag, 2001. LNCS 2102.

[9] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing surveys*, 24(3):293–318, 1992.

[10] J. Burch, E. Clarke, K. McMillan, D. Dill, and L.Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*. Springer-Verlag, May 1981. LNCS 131.

[13] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of 13th International Conference on Computer Aided Verification, (CAV'2001)*, pages 436–453. Springer-Verlag, 2001. LNCS 2102.

[14] M. Dam. Fixpoints of Büchi automata. In *Proceedings of the 12th International Conference of Foundations of Software Technology and Theoretical Computer Science*, pages 39–50. Springer-Verlag, 1992. LNCS 652.

[15] J. Desel and W. Reisig. Place/Transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, pages 122–173. Springer-Verlag, 1998. LNCS 1491.

[16] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.

[17] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[18] J. Esparza. Decidability and complexity of Petri net problems – An introduction. In *Lectures on Petri Nets I: Basic Models*, pages 374–428. Springer-Verlag, 1998. LNCS 1491.

[19] J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, pages 475–486. Springer-Verlag, July 2000. LNCS 1853.

[20] J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. Research Report A60, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, April 2000.

[21] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, pages 37–56. Springer-Verlag, May 2001. LNCS 2057.

[22] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. Research Report A68, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2001.

[23] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, pages 2–20. Springer-Verlag, 1999. LNCS 1664.

[24] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 87–106. Springer-Verlag, 1996. LNCS 1055.

[25] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm, 2001. Accepted for publication in Formal Methods for System Design.

[26] J. Esparza and C. Schröter. Reachability analysis using net unfoldings. In *Proceeding of the Workshop Concurrency, Specification & Programming 2000, volume II of Informatik-Bericht 140*, pages 255–270. Humboldt-Universität zu Berlin, 2000.

[27] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 1990.

[28] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[29] H. J. Genrich, K. Lautenbach, and P. S. Thiagarajan. Elements of general net theory. In *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory of Processes and Systems*, pages 21–163. Springer-Verlag, 1980. LNCS 84.

[30] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* Springer-Verlag, January 1996. LNCS 1032.

[31] U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57(2/3):125–147, 1983.

[32] B. Graves. Computing reachability properties hidden in finite net unfoldings. In *Proceedings of 17th Foundations of Software Technology and Theoretical Computer Science Conference*, pages 327–341. Springer-Verlag, 1997. LNCS 1346.

[33] K. Heljanko. Deadlock checking for complete finite prefixes using logic programs with stable model semantics (Extended abstract). In *Proceedings of the Workshop Concurrency, Specification & Programming 1998*, Informatik-Bericht Nr. 110, pages 106–115. Humboldt-University, Berlin, September 1998.

[34] K. Heljanko. Deadlock and reachability checking with finite complete prefixes. Research Report A56, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999. Licentiate's Thesis.

[35] K. Heljanko. Minimizing finite complete prefixes. In *Proceedings of the Workshop Concurrency, Specification & Programming 1999*, pages 83–95. Warsaw University, Warsaw, Poland, September 1999.

[36] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.

[37] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proceedings of 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 240–254. Springer-Verlag, 1999. LNCS 1579.

[38] K. Heljanko. Model checking with finite complete prefixes is PSPACE-complete. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'2000)*, pages 108–122. Springer-Verlag, August 2000. LNCS 1877.

[39] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, pages 218–232. Springer-Verlag, August 2001. LNCS 2154.

[40] K. Heljanko, V. Khomenko, and M. Koutny. Parallelisation of the Petri net unfolding algorithm. Technical Report CS-TR-733, Department of Computer Science, University of Newcastle upon Tyne, June 2001.

[41] K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 90–96. AAAI Press, Technical Report SS-01-01, March 2001.

[42] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, pages 200–212. Springer-Verlag, September 2001. LNAI 2173.

[43] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[44] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[45] ITU-T. *CCITT Specification and Description Language (SDL)*, Technical report Z.100 (1993), ITU-T, 1994.

[46] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Computational Logic – CL 2000; First International Conference*, pages 553–567. Springer-Verlag, 2000. LNAI 1861.

[47] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press / MIT Press, 1996.

[48] V. Khomenko and M. Koutny. LP deadlock checking using partial order dependencies. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'2000)*, pages 410–425. Springer-Verlag, August 2000. LNCS 1877.

[49] V. Khomenko and M. Koutny. Verification of bounded Petri nets using integer programming. Technical Report CS-TR-711, Department of Computer Science, University of Newcastle upon Type, 2000.

[50] V. Khomenko and M. Koutny. Towards an efficient algorithm for unfolding Petri nets. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, pages 366–380. Springer-Verlag, August 2001. LNCS 2154.

[51] O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Technion, Israel Institute of Technology, Haifa, Israel, June 1995. Also available as Technical Report, Technion, Computer Science Department, PhD Thesis PHD-1995-02.

[52] R. Langerak and E. Brinksma. A complete finite prefix for process algebra. In *Proceeding of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 184–195. Springer-Verlag, 1999. LNCS 1663.

[53] M. Mäkelä. A reachability analyser for algebraic system nets. Research Report A69, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, June 2001.

[54] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[55] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[56] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceedings of 9th International Conference on Computer Aided Verification (CAV'97)*, pages 352–363. Springer-Verlag, 1997. LNCS 1254.

[57] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[58] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1980.

[59] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[60] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429. Springer-Verlag, July 1997.

[61] I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, chapter 21, pages 491–521. Kluwer Academic Publishers, 2000.

[62] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[63] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[64] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1981.

[65] S. Römer. *Theorie und Praxis der Netzentfaltungen als Basis für die Verifikation nebenläufiger Systeme*. PhD thesis, Technische Universität München, Fakultät für Informatik, München, Germany, 2000.

[66] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[67] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report A47, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1997. Licentiate's thesis.

[68] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, April 2000.

[69] H. Tauriainen and K. Heljanko. Testing SPIN's LTL formula conversion into Büchi automata with randomly generated input. In *Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN'2000)*, pages 54–72. Springer-Verlag, 2000. LNCS 1885.

[70] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, pages 429–528. Springer-Verlag, 1998. LNCS 1491.

[71] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[72] K. Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, Digital Systems Laboratory, May 1998.

[73] K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 472–475. Springer-Verlag, June 1997. LNCS 1254.

[74] W. Vogler, V. Khomenko, and M. Koutny. Canonical prefixes of Petri net unfoldings. Technical Report CS-TR-741, Department of Computer Science, University of Newcastle upon Type, 2001.

[75] F. Wallner. Model checking LTL using net unfoldings. In *Proceeding of 10th International Conference on Computer Aided Verification (CAV'98)*, pages 207–218. Springer-Verlag, 1998. LNCS 1427.

[76] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proceedings of 12th International Conference on Computer Aided Verification, (CAV'2000)*, pages 124–138. Springer-Verlag, 2000. LNCS 1855.

ISBN 951-22-5886-2
ISSN 1457-7615