Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 56
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 56
Espoo 1999 HUT-TCS-A56

DEADLOCK AND REACHABILITY CHECKING WITH FINITE COMPLETE PREFIXES

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 56
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 56
Espoo 1999 HUT-TCS-A56

DEADLOCK AND REACHABILITY CHECKING WITH FINITE COMPLETE PREFIXES

Keijo Heljanko

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 951-22-4884-0
ISSN 0783-5396

Picaset Oy
Helsinki 1999

ABSTRACT: McMillan has presented a verification method for finite-state Petri nets based on finite complete prefixes of net unfoldings. Computational complexity of using finite complete prefixes as a symbolic representation of the state space is discussed. In addition a novel way of deadlock and reachability checking using the finite complete prefix approach is devised.

More specifically, the main contributions are: (i) The possible extensions calculation subroutine of the prefix generation algorithm is proved NP-complete. (ii) Model checking a fixed size CTL formula with finite complete prefixes is proved PSPACE-complete. (iii) The translations of the problems of deadlock and reachability checking using finite complete prefixes into the problem of finding a stable model of a logic program are devised. (iv) The implementation of the above mentioned translations in the `mcsmodels` tool is presented, with experimental results supporting the feasibility of the approach.

The implementation combines the prefix generator of the PEP-tool, the translations, and an implementation of a constraint-based logic programming framework, the `smodels` system. The experiments show that the proposed approach is quite competitive when compared to previous prefix based deadlock checking algorithms.

KEYWORDS: Computer aided verification, Petri nets, finite complete prefixes, net unfoldings, model checking, deadlocks, reachability, logic programs, stable model semantics, computational complexity

Contents

1 PREFACE

The author would like to thank Prof. Javier Esparza for the possibility of visiting his research group and for interesting discussions about the net unfolding method. Thanks also to Prof. Leo Ojala for the good research environment at TCS, and to Ilkka Niemelä and Tommi Junttila for many constructive ideas for this work. The tool `smodels` was programmed by Patrik Simons, who gave valuable support on its usage. Stephan Melzer and Stefan Römer provided the example nets, and also Linux binaries for ERVunfold and McMillan's algorithms, which both were invaluable. Thanks to Claus Schröter for supplying markings for reachability checking experiments, and to Burkhard Graves and Bernd Grahlmann for supplying C source code to read PEP prefix files.

The financial support of Helsinki Graduate School in Computer Science and Engineering (HeCSE), the Academy of Finland (Project 8309), the Support Foundation of Helsinki University of Technology, and the Emil Aaltonen Foundation are gratefully acknowledged.

Last but not least I would like to thank my girlfriend Virpi Leván for her loving support and encouragement.

Keijo Heljanko
Espoo, Finland, December 1999

2 INTRODUCTION

Concurrent and distributed systems are becoming into more widespread use. They are used to implement, for example, distributed data communication systems such as telecommunication networks, safety critical systems such as railroad signaling systems, and hardware systems such as parallel computers. However, the concurrency in these systems has made them more difficult to design and to test. Some errors in the system might show up only under certain timing conditions, which makes such errors hard to find and reproduce.

Computer aided verification has given us new tools to find such errors, one of these tools is *model checking*. In model checking a formal model of the behavior of the system is derived from the system description. Interesting system properties are specified by *temporal logic*, and a model checking algorithm is then used to check using the formal model whether the system meets its specification. A main feature of model checking is that it can be mostly automated, which means it can also be used by non-expert users. The main industrial use of model checking has so far been debugging. When the specified property does not hold, the model checker can generate a counterexample execution. Such a counterexample can be then used to help in locating the possible fault in the system.

One of the main problems in applying the computer aided verification methods, such as model checking, to larger systems is the so called *state explosion problem*. For an overview of the problem and some of the methods for alleviating it see e.g. [40]. One of the main causes of the state explosion problem is that the number of reachable states of a system can be exponential in the number of concurrent components of the system. Therefore the approach to verify a property by simply generating the state space and then investigating whether the property holds can end up using too much memory.

There have been many different approaches to alleviate this problem. For example, the stubborn set, persistent set and sleep set methods all use knowledge of the structure of the system to reduce the number of states that need to be explored to verify a property. This structural information can be used to only generate a subset of the states of the interleaved state space.

Quite another approach is to use symbolic methods. There instead of explicitly storing each of the states of the state space, one stores a symbolic representation of the state space. The most well known of this class of methods is the use of binary decision diagrams (BDDs), see e.g. [29]. The BDD method seems to be best suited for synchronous hardware verification, and has in this domain had quite a good success.

In this work we consider *net unfoldings*, another symbolic technique to alleviate the state explosion problem. The approach is based on the concept of a net unfolding, a partial order semantics of Petri nets introduced in [33], and later described in more detail in [6] under the name of *branching processes*. The unfolding of a Petri net is another net, usually infinite but with a simpler structure. McMillan was the first to propose a practical verification algorithm for finite state Petri nets based on net unfoldings in his seminal paper [28, 30]. The basic idea is to construct a finite initial part of the unfolding, which contains all information about the state space. We call such an initial part a *finite complete prefix*. McMillan's algorithm created prefixes which could sometimes be exponentially larger than the interleaved state space. This problem was fixed for 1-safe Petri nets by Esparza, Römer and Vogler in [11, 12]. The finite complete prefix approach is not Petri net specific, but can be applied also to other models of concurrency where the notion of independence between events can be defined. Such definitions for other formalisms can be found in [10, 25].

A finite complete prefix can thus be seen as a symbolic representation of the state space. In addition to containing all the information about the state space, it also has all the partial order information contained in the unfolding. Unlike the BDD approach, the net unfolding approach is best suited for asynchronous systems. The net unfolding approach is similar to the symbolic model checking without BDDs approach [2] in the way it uses NP-solvers to solve reachability problems on the symbolic representation. However, this approach is mainly used for falsification, because unlike net unfoldings, the creation of the symbolic representation is simpler and does not guarantee that it contains all the reachable states. Of course, the completeness of the symbolic representation can be checked afterwards as suggested in [2], but this has at least currently been deemed too computationally expensive in practice. The net unfolding method can also easily be made into an incomplete falsification method generating just a finite prefix instead of an also complete one.

The net unfolding approach has had several applications. McMillan devised a deadlock checking algorithm using a branch-and-bound algorithm [28, 30]. Later Melzer and Römer improved on his results in [32] by using a mixed integer programming approach. In this work we have devised a deadlock and reachability checker, using a translation into the problem of finding a stable model of a logic program. Our results show that suggested approach is quite competitive when compared to the approaches of McMillan and Melzer/Römer. The main results have been published in [18, 20, 21].

The net unfolding approach has also been used for checking properties of speed independent circuits, see [23, 24, 31]. It was also used for model checking a simple branching time logic in [7], see, however, also [16] for a description of a problem with this approach and an approach how to fix it. There is also an algorithm for LTL model checking using prefixes created by Wallner, however, the paper describing an earlier version [43] has a bug in it (F. Wallner, private communication). There are several different approaches to do temporal logic model checking using prefixes, and the trade-offs of different approaches are not yet well-understood.

In this work we will discuss both the theoretical and practical aspects of using finite complete prefixes as a symbolic representation of the reachability graph. More precisely we discuss the finite complete prefix generation, complexity of model checking using prefixes, and devise a method and implementation for deadlock and reachability checking with finite complete prefixes.

The main contributions of this work are: (i) The possible extensions calculation subroutine of the prefix generation algorithm is proved NP-complete. (ii) Model checking a fixed size CTL formula with finite complete prefixes is proved PSPACE-complete. (iii) The translations of the problems of deadlock and reachability checking using finite complete prefixes into the problem of finding a stable model of a logic program are devised. (iv) The implementation of the above mentioned translations in the `mcsmodels` tool is presented, with experimental results supporting the feasibility of the approach.

The rest of this work is structured as follows. First in Chapter 3 we present the Petri net notations used in the paper. In Chapter 4 an algorithm for constructing a finite complete prefix is given, together with complexity analysis of the possible extensions subroutine. Next in Chapter 5 we discuss the complexity of model checking with finite complete prefixes.

Chapter 6 gives the necessary background to the stable model semantics of logic programs to follow the translations of deadlock and reachability properties with prefixes given in Chapter 7. Chapter 8 discusses implementation of these translations and the optimizations used, and gives experimental results. In Chapter 9 we conclude and discuss directions for future research.

3 PETRI NET DEFINITIONS

First we define basic Petri net notations. Next we introduce *occurrence nets*, which are 1-safe Petri nets of a restricted form. Then *branching processes* are given as a way of describing partial order semantics for Petri nets. Last but not least we define *finite complete prefixes* as a way of giving a finite representation of this partial order behavior. We follow mainly the notation of [11, 12, 32].

3.1 PETRI NETS

A triple $\langle S, T, F\rangle$ is a *net* if $S \cap T = \emptyset$ and $F \subseteq (S \times T) \cup (T \times S)$. The elements of S are called *places*, and the elements of T *transitions*. Places and transitions are also called *nodes*. We identify F with its characteristic function on the set $(S \times T) \cup (T \times S)$. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in S \cup T \mid F(y, x) = 1\}$. The *postset* of a node x, denoted by $x^\bullet$, is the set $\{y \in S \cup T \mid F(x, y) = 1\}$. Their generalizations on sets of nodes $X \subseteq S \cup T$ are defined as ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$, and $X^\bullet = \bigcup_{x \in X} x^\bullet$, respectively.

A *marking* of a net $\langle S, T, F\rangle$ is a mapping $S \mapsto \mathbb{N}$. A marking M is identified with the multi-set which contains $M(s)$ copies of s for every $s \in S$. A 4-tuple $\Sigma = \langle S, T, F, M_0\rangle$ is a *net system* if $\langle S, T, F\rangle$ is a net and M_0 is a marking of $\langle S, T, F\rangle$. A marking M enables a transition t if $\forall s \in S : F(s, t) \leq M(s)$. If t is enabled, it can *occur* leading to a new marking (denoted $M \xrightarrow{t} M'$), where M' is defined by $\forall s \in S : M'(s) = M(s) - F(s, t) + F(t, s)$. A marking M is a *deadlock marking* if no transition t is enabled by M. A marking M_n is *reachable* in Σ if there is an *execution*, i.e. a sequence of transitions $t_1, t_2, \ldots, t_n$ and markings $M_1, M_2, \ldots, M_{n-1}$ such that: $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots M_{n-1} \xrightarrow{t_n} M_n$. A reachable marking is 1-safe if $\forall s \in S : M(s) \leq 1$. A net system Σ is 1-safe if all its reachable markings are 1-safe. In this work we will restrict ourselves to net systems which are 1-safe, have a finite number of places and transitions, and also in which each transition $t \in T$ has both nonempty pre- and postsets.

3.2 OCCURRENCE NETS

We use $<_F$ $(\leq_F)$ to denote the (reflexive) transitive closure of F. Define $\Sigma = \langle S, T, F\rangle$ to be a net and let $x_1, x_2 \in S \cup T$. The nodes x_1 and x_2 are in *conflict*, denoted by $x_1 \# x_2$, if there exist $t_1, t_2 \in T$ such that $t_1 \neq t_2$, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, $t_1 \leq_F x_1$, and $t_2 \leq_F x_2$. The nodes x_1 and x_2 are *concurrent*, denoted by $x_1 \mathit{co}\, x_2$, if neither $x_1 <_F x_2$ nor $x_2 <_F x_1$ nor $x_1 \# x_2$.

An occurrence net is a net $N = \langle B, E, F\rangle$ such that:

- $\forall b \in B : |{}^\bullet b| \leq 1$,
- F is acyclic, i.e. the irreflexive transitive closure of F is a partial order,
- N is finitely preceded, i.e. for any node x of the net, the set of nodes y

such that $y \leq_F x$ is finite, and

- $\forall x \in S \cup T : \neg(x \# x)$.

The elements of B and E are called *conditions* and *events*, respectively. The set $Min(N)$ denotes the set of minimal elements of the transitive closure of F. In this work the minimal elements will all be conditions, and thus the set $Min(N)$ can be intuitively seen as an initial marking of a 1-safe net system, which has the net structure of the occurrence net N.

A *configuration* C of an occurrence net is a set of events satisfying:

- If $e \in C$ then $\forall e' \in E : e' \leq_F e$ implies $e' \in C$ (C is causally closed),
- $\forall e, e' \in C : \neg(e \# e')$ (C is conflict-free).

A configuration is a set of events which can be fired in some (not necessarily unique) order starting from the marking $Min(N)$ of the occurrence net. A configuration thus represents a set of enabled sequences of transitions in the initial marking $Min(N)$.

A set of conditions is a *co-set* if all its elements are pairwise in *co* relation. A maximal co-set with respect to set inclusion is called a *cut*. Given a configuration of an occurrence net, we can define what we mean by the notion "the marking of the occurrence net reached after firing all the events in the configuration". The co-set of a configuration $Cut(C) \subseteq B$, is a cut: $Cut(C) = (Min(N) \cup C^\bullet) \setminus {}^\bullet C$. A configuration C is a *deadlock configuration* if the set $Cut(C)$ does not enable any event $e \in E$.

A *local configuration* $[e]$ of an event e is the set of events e', such that $e' \leq_F e$. A local configuration is the (unique) smallest set of events which must have been fired when the event e has been fired.

3.3 BRANCHING PROCESSES

Branching processes are "unfoldings" of net systems and were introduced by Engelfriet [6]. Let $N_1 = \langle S_1, T_1, F_1 \rangle$ and $N_2 = \langle S_2, T_2, F_2 \rangle$ be two nets. A *homomorphism* h is a mapping $S_1 \cup T_1 \mapsto S_2 \cup T_2$ such that: $h(S_1) \subseteq S_2 \wedge h(T_1) \subseteq T_2$, and for all $t \in T_1$, the restriction of h to ${}^\bullet t$ is a bijection between ${}^\bullet t$ and ${}^\bullet h(t)$, and similarly for $t^\bullet$ and $h(t)^\bullet$. A *branching process* of a net system Σ is a tuple $\beta = \langle N', h \rangle$, where N' is a occurrence net, and h is a homomorphism from N' to $\langle S, T, F \rangle$ such that: the restriction of h to $Min(N')$ is a bijection between $Min(N')$ and M_0, and $\forall e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2 \wedge h(e_1) = h(e_2)$ then $e_1 = e_2$. Thus the homomorphism h maps the conditions and events of an occurrence net to the places and transitions of the corresponding net system in a way, which respects both the initial marking and the labeling of all the transitions and their pre- and postsets. It also disallows duplicating events which have the same label and preset.

The set of places associated with a configuration C of β is denoted by $Mark(C) = h(Cut(C))$. Thus a configuration of the branching process always corresponds to some (reachable) marking of the original net system. It is shown in [6] that a net has a maximal branching process up to isomorphism, called the *unfolding* of the system.

Given a configuration C of the unfolding, we denote by $C \oplus E$ the set $C \cup E$ (the *extension* of C by E), under the condition that $C \cup E$ is a configuration of the unfolding satisfying $C \cap E = \emptyset$. The possible extensions of a configuration correspond to sets of events which can be fired in the occurrence net starting from the marking $Cut(C)$.

Given a configuration C of a branching process $\beta = \langle N, h \rangle$, we define the *future of a configuration*, denoted $\Uparrow C$, as the pair $\langle N', h' \rangle$, where N' is the unique subnet of N whose set of nodes is $\{x \mid x \notin C \wedge \forall y \in C : \neg(x \# y)\}$, and h' is the restriction of h to the nodes of N'. In other words $\Uparrow C$ contains the part of the unfolding "laying after" the configuration C.

4 ALGORITHM FOR CONSTRUCTING A FINITE COMPLETE PREFIX

McMillan was the first to present an algorithm to construct a finite complete prefix from a given finite-state Petri net [28, 30]. Later Esparza, Römer and Vogler improved the algorithm [11, 12] of McMillan, which could sometimes create a finite complete prefix exponentially larger than the interleaved state space of the system. In this work we restrict ourselves to 1-safe Petri nets. The approach of McMillan works also for non-1-safe Petri nets with finite state space. However, the original improvement of Esparza et.al. [11] works only for 1-safe Petri nets. To obtain a similar bound on n-safe nets, more changes to the approach are needed [12, 17].

Most of the material of this chapter is from [11, 12], however, the analysis of the possible extensions calculation including the complexity result of Theorem 10 are new. We will skip those proofs of theorems which are presented in [11, 12]. First we give an algorithm to construct the unfolding of a net system. Later this algorithm is refined to construct a finite complete prefix contained in the unfolding.

4.1 CONSTRUCTING THE UNFOLDING

We first define a data structure for the representation of branching processes. A branching process of a net system Σ is implemented as a set $\{n_1, \ldots, n_k\}$ of *nodes*. A node is either a condition or an event. A *condition* is a record that contains two fields: a *place* of Σ, and a pointer to the unique input event of that condition, or NIL, in case the condition has an empty preset. In the pseudo-code the conditions are represented as a pair (s, e) or $(s, \emptyset)$. An *event* is also a record with two fields: a *transition* of Σ, and a list of pointers to the input conditions of that event. In pseudo-code an event is represented as a pair (t, X).

Notice that the flow relation of the branching process, as well as the labeling function h are encoded by the set of nodes. Thus a set of nodes uniquely defines (up to isomorphism) a branching process. To define the unfolding process we need a notion of "events that can be added to a given branching process":

Definition 1 *Let t be a transition of Σ with output places $s_1, \ldots, s_n$. A pair $e = (t, X)$ is a* possible extension *of a branching process $\{n_1, \ldots, n_k\}$ if $\{n_1, \ldots, n_k, e, (s_1, e), \ldots, (s_n, e)\}$ is also a branching process.*

We denote by $PE(\beta)$ the set of all possible extensions of a branching process β. Another way of characterizing the possible extensions is the following:

Proposition 2 *Let β be a branching process of a net system Σ. The* possible extensions *of β are the pairs (t, X), where X is a co-set of conditions of β and t is a transition of Σ such that*

- *$h(X) = {}^\bullet t$, and*
- *(t, X) does not already belong to β.*

4.2 THE UNFOLDING ALGORITHM

Now we are ready to present the unfolding algorithm. As can be seen from the Algorithm 1, it uses the possible extensions calculation as its innermost subroutine.

Algorithm 1 *The unfolding algorithm*

input: A net system $\Sigma = \langle S, T, F, M_0 \rangle$, where $M_0 = \{s1, \ldots, s_n\}$.
output: The unfolding Unf of Σ.
begin
$Unf := \{(s_1, \emptyset), \ldots, (s_n, \emptyset)\}$;
$pe := PE(Unf)$;
while $pe \neq \emptyset$ **do**
    append to Unf an event $e = (t, X)$ of pe and a condition (s, e)
        for every output place s of t;
    $pe := PE(Unf)$;
endwhile
end

As an example of the unfolding, look at the 1-safe net system Σ in Figure 4.1. The initial part of the (infinite) unfolding of Σ is presented in Figure 4.2. The homomorphism h is given by the labels in the parentheses.

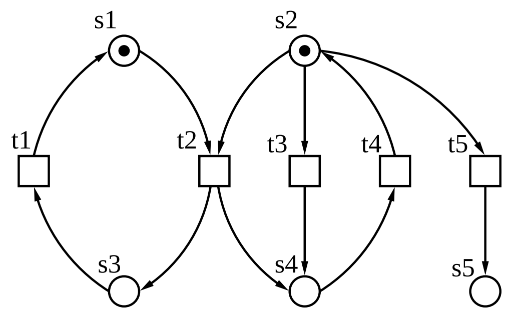


Figure 4.1: A 1-safe net system Σ.

Notice that the algorithm will terminate only when the input net system has no infinite execution sequences. It is also correct only under the fairness assumption that every event added to pe is eventually chosen to extend Unf. If no fairness is assumed, it might be that there is some infinite execution of the unfolding which is expanded forever, and thus some other part of the unfolding might not become generated at all. Under the fairness assumption we know that if a event appears as a possible extension, it will also appear at some point in the unfolding (i.e. no "branch of the unfolding" is ignored forever). One can implement such a fairness notion by, for example, adding the events to the unfolding in a non-decreasing local configuration order, in which case the Algorithm 1 becomes a breath-first search type algorithm.

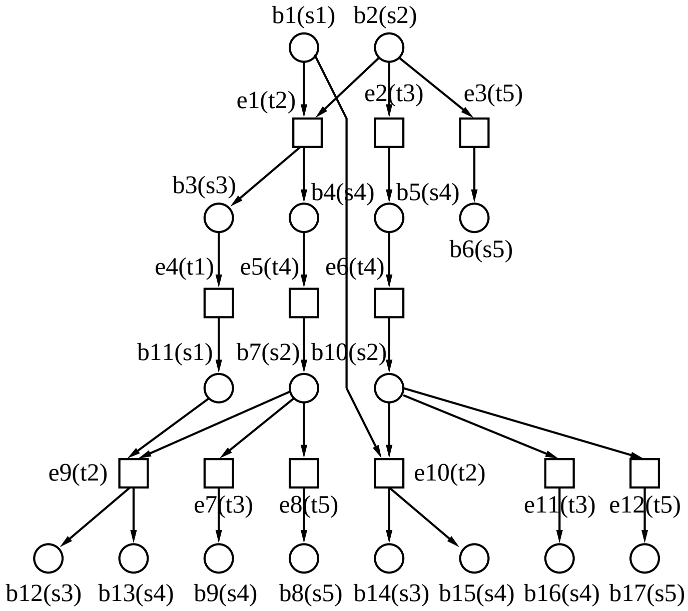


Figure 4.2: An initial part of the unfolding of Σ.

4.3 CONSTRUCTING A FINITE COMPLETE PREFIX

Let's first define what it means for a prefix to be complete.

Definition 3 *A finite 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 $h(e) = t$.*

The unfolding of a net system is always complete. Intuitively a prefix of the unfolding is complete when it contains as much information as the unfolding, in the sense that the unfolding can be reconstructed from it. Because the net systems we consider in this work have only a finite number of reachable markings, the unfolding always contains at least one *finite complete prefix*. The idea of McMillan is to identify some events, called cut-off events, at which the construction of an unfolding can be stopped without losing information. The intuition behind cutoff events is that for each cut-off event e there already exists another event e' "earlier" in the prefix. The markings reachable after executing e can also be reached after executing e', and thus the markings after e need not to be considered any further. To formalize this idea we present the following proposition.

Proposition 4 *If β is a branching process of $\Sigma = \langle S, T, F, M_0 \rangle$ and C is a configuration of β, then $\Uparrow C$ is a branching process of the net system $\Sigma' = \langle S, T, F, Mark(C) \rangle$. Moreover, if β is the unfolding of Σ, then $\Uparrow C$ is the unfolding of Σ' (up to isomorphism).*

Now let C_1 and C_2 be two finite configurations with the same marking, i.e. $\mathit{Mark}(C_1) = M = \mathit{Mark}(C_2)$. By Proposition 4: $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic to the unfolding of $\langle S, T, F, M\rangle$, and therefore also isomorphic to each other. Let I_1^2 be an isomorphism between $\Uparrow C_1$ and $\Uparrow C_2$. The isomorphism induces a mapping from the finite extensions of C_1 onto the finite extensions of C_2: It maps $C_1 \oplus E$ onto $C_2 \oplus I_1^2(E)$.

Basically what Proposition 4 gives us is the fact that when two configurations have the same marking, their "futures" ($\Uparrow C_1$ and $\Uparrow C_2$) are isomorphic. Therefore, if we have already created, for example, the future of C_1, then the future of C_2 needs not to be generated, as all the information about it can be obtained by looking at the future of C_1. The finite complete prefix algorithm uses such information to avoid generating isomorphic futures of configurations.

We need to define what it means for a configuration to be "earlier" or "smaller" than some other configuration. This is needed to decide when to safely stop the generation of a prefix.

Definition 5 *A partial order $\prec$ on the 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 I_1^2 from above satisfies $C_1 \oplus E \prec C_2 \oplus I_1^2(E)$ for all finite extensions of C_1.*

McMillan used originally the following order: $C_1 \prec C_2$ if $|C_1| < |C_2|$. It is easy to prove that McMillan's order is adequate, see [11, 12]. However, it is not total when restricted to configurations having the same marking, i.e. $C_1 \neq C_2 \wedge \mathit{Mark}(C_1) = \mathit{Mark}(C_2)$ does not imply $C_1 \prec C_2 \vee C_2 \prec C_1$. This problem manifests in a way in which the finite complete prefix might end up being exponentially larger than necessary. Esparza, Römer and Vogler fixed this problem by given a more involved definition of an adequate order $\prec$, which fulfills the condition above [11, 12]. (In fact, the order they define is stronger, it is total on all configurations, i.e. $C_1 \neq C_2$ implies $C_1 \prec C_2 \vee C_2 \prec C_1$. However, such totality on all configurations is not needed by the finite complete prefix generation algorithms.)

Definition 6 *Let $\prec$ be an adequate order on the configurations of the unfolding of a net system. Let β be a prefix of the unfolding containing an event e. The event e is a* cut-off event *of β (with respect to $\prec$) if β contains a local configuration $[e']$ such that*

- *$\mathit{Mark}([e']) = \mathit{Mark}([e])$, and*
- *$[e'] \prec [e]$.*

Now we can define an algorithm to compute a finite complete prefix, Algorithm 2. This algorithm has as a parameter an adequate order $\prec$. Each

Algorithm 2 *The finite complete prefix algorithm*

input: A n-safe net system $\Sigma = \langle S, T, F, M_0 \rangle$, where $M_0 = \{s1, \ldots, s_n\}$.
output: A finite complete prefix Fin of Unf.
begin
$Fin := \{(s_1, \emptyset), \ldots, (s_n, \emptyset)\}$; *cut-off* $:= \emptyset$;
$pe := PE(Fin)$;
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 (s, e)
      for every output place s of t;
    $pe := PE(Fin)$;
    if e is a cut-off event for Fin **then** *cut-off* $:=$ *cut-off* $\cup \{e\}$;
  else
    $pe := pe \setminus \{e\}$;
  endif
endwhile
end

order can lead to a different prefix being generated. We will denote by $CutOffs(\beta)$ the set *cut-off* generated by the finite complete prefix algorithm for the prefix $\beta = Fin$.

See Figure 4.3 for a finite complete prefix of the 1-safe Petri net system Σ in Figure 4.1. The cut-off events are marked with crosses in the prefix.

An interesting fact is that Algorithm 2 will never create a finite complete prefix which has more non-cut-off events as there are reachable markings of the net system Σ, when using the adequate order of Esparza et.al. [11, 12]. For a correctness proof and more information about the algorithm, see [11, 12].

4.4 COMPLEXITY OF POSSIBLE EXTENSIONS CALCULATION

The possible extensions calculation is the innermost loop of all known prefix generation algorithms. Therefore its computational complexity is of interest to us. To investigate it, we define the following decision problem:

Definition 7 *POSSIBLE-EXTENSION:*
Given a net system Σ and a finite prefix β of its unfolding, decide whether there exists a transition $t \in T$ and a set of conditions $X \subseteq B$, such that $(t, X) \in PE(\beta)$.

Notice that we are given as input both the net system Σ and the prefix β, and thus the complexity results we show are in respect to the joint size of Σ and β. We define the joint size of the prefix β and the corresponding net system Σ to be the sum of the lengths of the binary encodings of $\beta = \langle \langle B, E, F \rangle, h \rangle$ and $\Sigma = \langle S, T, F', M_0 \rangle$. We define NP to be the complexity class of languages decided by nondeterministic Turing machines in

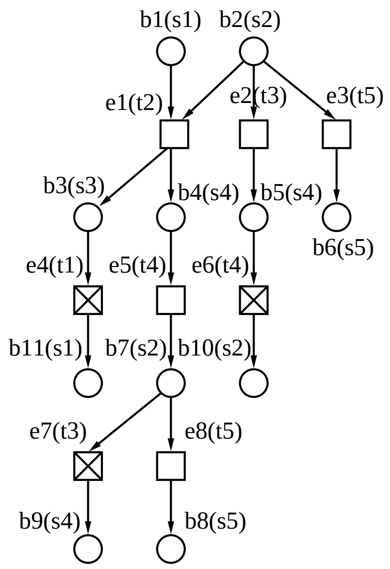


Figure 4.3: A finite complete prefix of Σ.

polynomial time. We also use the standard notations of a problem being in a complexity class, being hard for a complexity class, and being complete for a complexity class. For an overview of these terms and computational complexity in general, see e.g. [36]. In this work we use as reductions *polynomial-time many-one reductions* (i.e. Karp reductions), instead of logarithmic-space reductions as in [36]. (For further information about this difference, see [36], page 177.)

Lemma 8 *POSSIBLE-EXTENSION is in NP.*

Proof:
We create a nondeterministic algorithm with a polynomial running time. The requirements for a pair (t, X) to be a possible extension of β are:

- $X \subseteq B$,
- X is a co-set, i.e. all events of X are pairwise in co-relation: $b_i, b_j \in X \wedge b_i \neq b_j$ implies:

  a) $\neg(b_i <_F b_j) \wedge$

  b) $\neg(b_j <_F b_i) \wedge$

  c) $\neg(b_i \# b_j)$,
- $t \in T$,
- $|^\bullet t| = |X|$,
- $^\bullet t = h(X)$, and
- there is no $e \in E$, such that $h(e) = t \wedge {}^\bullet e = X$.

Algorithm 3 *The POSSIBLE-EXTENSION algorithm*

```
input: A net system Σ = ⟨S, T, F′, M₀⟩, its finite prefix β = ⟨⟨B, E, F⟩, h⟩.
output: Accept/reject.
begin
Guess a set X ⊆ B, where X = {b₁, ... , bₙ};
if n ≥ 2 then
    for i := 1 to n-1 do
        for j := i+1 to n do
            if REACHABLE(bᵢ, bⱼ) then reject; /* a) */
            if REACHABLE(bⱼ, bᵢ) then reject; /* b) */
            for b ∈ B do
                if |b•| < 2 then continue;
                for e₁ ∈ b• do
                    if REACHABLE(e₁, bᵢ) then
                        for e₂ ∈ (b• \ {e₁}) do
                            if REACHABLE(e₂, bⱼ) then
                                reject; /* c) */
                        endfor
                    endif
                endfor
            endfor
        endfor
    endfor
endif
for t ∈ T do
    if |•t| ≠ |X| ∨ •t ≠ h(X) then continue;
    found := false;
    for e ∈ E do
        if h(e) = t ∧ •e = X then found := true;
    endfor
    if found = false then accept; /* output (t, X); */
endfor
reject;
end
```

The nondeterministic algorithm, Algorithm 3, uses the definition above to accept iff a pair (t, X) fulfilling all the conditions above exists. The algorithm is quite straightforward and proceeds in the same order as the list of requirements above. The program first guesses a subset X of the set B. Thus the amount of nondeterminism needed is polynomial in the size of the input.

Then the algorithm runs a checking phase, which accepts iff there exists a pair (t, X) which fulfills all the conditions listed above. All the loops of the program have a polynomial upper bound on the number of iterations in the size of the input, and thus the program has a polynomial running time after the nondeterministic initial guess has been made.

To simplify the presentation, we use a subroutine $REACHABLE(x_1, x_2)$. It returns $true$, if the node x_2 can be reached from the node x_1 using the arcs of the flow relation F of the prefix, and *false* otherwise. Therefore $REACHABLE(x_1, x_2)$ return *true* exactly when $x_1 <_F x_2$. It is easy to see that this subroutine can be implemented in polynomial time.

The only nontrivial part is checking for the set to be a co-set. This is done by implementing an algorithmic version of the definition of a co-set. The algorithm loops over all pairs of conditions (b_i, b_j) of the set X, and checks the three co-set requirements. The requirements a and b can be checked by simple reachability checks. For the requirement c we use the definition of the conflict relation. The requirement c is violated if there exist $b \in B$, $e_1, e_2 \in b^\bullet$ such that $e_1 \neq e_2$, $e_1 <_F b_i$, and $e_2 <_F b_j$.

Thus Algorithm 3 accepts iff there exists a possible extension (t, X) of β. Additionally, Algorithm 3 is in NP in the joint size of β and Σ. □

Algorithm 3 is a decision algorithm and cannot thus be directly used to compute the set $PE(\beta)$. However, it can be turned into a deterministic algorithm with exponential running time in the number of conditions, which would output all the possible extensions. Notice, however, that the algorithm was created for simplicity of description, and is thus not an optimal one to compute the set of possible extensions.

Lemma 9 *POSSIBLE-EXTENSION is NP-hard.*

The reader is suggested to look at the example following the proof, especially Figures 4.4 and 4.5 will be useful for visualizing the reduction we use.

Proof:
By a polynomial-time many-one reduction from the NP-complete problem 3SAT into the problem POSSIBLE-EXTENSION. The problem 3SAT is an instance of the Boolean satisfiability problem, in which the Boolean expression is in conjunctive normal form, with each conjunct having exactly 3 literals, see e.g. [36].

We are given an instance ϕ of 3SAT with m clauses $C_1, \ldots, C_m$, with each clause being $C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$, with α_{ij} being either a Boolean variable from the set $\{x_1, \ldots, x_n\}$, or a negation of a variable $\{\neg x_1, \ldots, \neg x_n\}$.

We reduce the 3SAT problem in several phases. First we define a net system $\Sigma = \langle S, T, F, M_0 \rangle$ as:

- $S = \{x_1, \ldots, x_n\} \cup$
  $\{px_{11}, \ldots, px_{1m}\} \cup \ldots \cup \{px_{n1}, \ldots, px_{nm}\} \cup$
  $\{nx_{11}, \ldots, nx_{1m}\} \cup \ldots \cup \{nx_{n1}, \ldots, nx_{nm}\} \cup$
  $\{m_1, \ldots, m_m\} \cup$
  $\{c_1, \ldots, c_m\} \cup$
  $\{s\}$,
- $T = \{tpx_1, \ldots, tpx_n\} \cup$
  $\{tnx_1, \ldots, tnx_n\} \cup$
  $\{ts_{ij} \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\} \cup$
  $\{t\}$,

- $F = \{\langle x_1, tpx_1\rangle, \ldots, \langle x_n, tpx_n\rangle\} \cup$
  $\{\langle x_1, tnx_1\rangle, \ldots, \langle x_n, tnx_n\rangle\} \cup$
  $\{\langle tpx_1, px_{11}\rangle, \ldots, \langle tpx_1, px_{1m}\rangle\} \cup \ldots \cup$
  $\{\langle tpx_n, px_{n1}\rangle, \ldots, \langle tpx_n, px_{nm}\rangle\} \cup$
  $\{\langle tnx_1, nx_{11}\rangle, \ldots, \langle tnx_1, nx_{1m}\rangle\} \cup \ldots \cup$
  $\{\langle tnx_n, nx_{n1}\rangle, \ldots, \langle tnx_n, nx_{nm}\rangle\} \cup$
  $\{\langle px_{ki}, ts_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\} \wedge k \in \{1, \ldots, n\} \wedge$
  $\alpha_{ij} = x_k\} \cup$
  $\{\langle nx_{ki}, ts_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\} \wedge k \in \{1, \ldots, n\} \wedge$
  $\alpha_{ij} = \neg x_k\} \cup$
  $\{\langle m_i, ts_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\} \cup$
  $\{\langle ts_{ij}, c_i\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\} \cup$
  $\{\langle c_1, t\rangle, \ldots, \langle c_m, t\rangle\} \cup$
  $\{\langle t, s\rangle\}$, and

- $M_0 = \{x_1, \ldots, x_n\} \cup$
  $\{m_1, \ldots, m_m\}$.

It is easy to see that the created net system is polynomial in the size of the input formula ϕ. It is also straightforward to generate it in polynomial time. The net system is also 1-safe, but this property is not needed in the proofs.

We now turn to the second part of the mapping. We need to create a prefix of the unfolding of the net system Σ. One such a prefix is the prefix $\beta' = \langle\langle B', E', F'\rangle, h'\rangle$:

- $B' = \{bx_1, \ldots, bx_n\} \cup$
  $\{bpx_{11}, \ldots, bpx_{1m}\} \cup \ldots \cup \{bpx_{n1}, \ldots, bpx_{nm}\} \cup$
  $\{bnx_{11}, \ldots, bnx_{1m}\} \cup \ldots \cup \{bnx_{n1}, \ldots, bnx_{nm}\} \cup$
  $\{bm_1, \ldots, bm_m\} \cup$
  $\{bc_{ij} \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\}$,

- $E' = \{epx_1, \ldots, epx_n\} \cup$
  $\{enx_1, \ldots, enx_n\} \cup$
  $\{es_{ij} \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\}$,

- $F' = \{\langle bx_1, epx_1\rangle, \ldots, \langle bx_n, epx_n\rangle\} \cup$
  $\{\langle bx_1, enx_1\rangle, \ldots, \langle bx_n, enx_n\rangle\} \cup$
  $\{\langle epx_1, bpx_{11}\rangle, \ldots, \langle epx_1, bpx_{1m}\rangle\} \cup \ldots \cup$
  $\{\langle epx_n, bpx_{n1}\rangle, \ldots, \langle epx_n, bpx_{nm}\rangle\} \cup$
  $\{\langle enx_1, bnx_{11}\rangle, \ldots, \langle enx_1, bnx_{1m}\rangle\} \cup \ldots \cup$
  $\{\langle enx_n, bnx_{n1}\rangle, \ldots, \langle enx_n, bnx_{nm}\rangle\} \cup$
  $\{\langle bpx_{ki}, es_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\} \wedge k \in \{1, \ldots, n\} \wedge$
  $\alpha_{ij} = x_k\} \cup$
  $\{\langle bnx_{ki}, es_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\} \wedge k \in \{1, \ldots, n\} \wedge$
  $\alpha_{ij} = \neg x_k\} \cup$
  $\{\langle bm_i, es_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\} \cup$
  $\{\langle es_{ij}, bc_{ij}\rangle \mid i \in \{1, \ldots, m\} \wedge j \in \{1, 2, 3\}\}$, and

- the mapping h' maps each event exy_{ij} to the transition txy_{ij}, and each condition bxy_{ij} to the place xy_{ij}, except for conditions bc_{ij}, which are mapped (three to one) to the place c_i.

The size of β' is polynomial in the size of the input. It can be generated by a similar polynomial time algorithm which created the net system Σ. It has three times as many conditions bc_{ij}, as there are places c_i in the net system Σ. However, it doesn't have any events corresponding to the transition t.

It is easy to see that β' is a prefix of the unfolding of Σ. One way to prove it is to create the net system Σ', in which transition t has been removed from Σ. Such a removal of transitions can never result in an unfolding containing more events, and thus the resulting unfolding is always a prefix of the original unfolding. We can now run the unfolding algorithm (Algorithm 1) on the net Σ', and then prove that the resulting unfolding is isomorphic to β'. (The net Σ' has been defined in such a way that it is almost, not counting places c_i, isomorphic to its unfolding.)

The next thing to prove is that if we have a satisfying truth assignment V for ϕ, which maps each variable x_i to either $true$ or $false$, then the prefix β' has an extension by the transition t. This proof is quite straightforward, just initialize the configuration C of the β' as follows:

- for all $i \in \{1, \ldots, n\}$, such that $V(x_i) = true \,:\, epx_i \in C$, and
- for all $i \in \{1, \ldots, n\}$, such that $V(x_i) = false \,:\, enx_i \in C$.

We now notice that because V is a satisfying truth assignment, for each clause C_i there exists some literal α_{ij}, where $j \in \{1, 2, 3\}$, such that the valuation of the variable x_k associated with the literal α_{ij} satisfies the clause C_i. We pick any such set of literals $\{\alpha_{1j_1}, \ldots, \alpha_{mj_m}\}$. It is now easy to see from the definition of C and the preset of the event es_{ij_i}, that $C \oplus \{es_{ij_i}\}$ is a configuration of β'. Also all such events es_{ij}, es_{kl} are all in pairwise *co* relation when $i \neq k$. Therefore we can extend the configuration C by adding m events to create the configuration $C' = C \oplus \{es_{1j_1}, \ldots, es_{mj_m}\}$. Now it is easy to check that the corresponding co-set $X = Cut(C')$ is a set of conditions, such that (t, X) is a possible extension of β'.

In the other direction we need to prove that if there exists a possible extension of β', then there exists some satisfying truth assignment V for ϕ. We need to consider only the possible extension by the transition t, because Σ has no new places compared to Σ', and thus has no new extensions by other transitions. Let X be the set of conditions corresponding to any possible extension (t, X), and let C be the corresponding configuration, such that $Cut(C) = X$. We construct a truth assignment V from C as follows. Let V be any total function which maps each of the variables $x_1, \ldots x_n$ to either $true$ or $false$ with the definition:

- for all $i \in \{1, \ldots, n\} \,:\,$ if $epx_i \in C$, then $V(x_i) = true$, else $V(x_i) = false$.

It is easy to see that C also has to contain m events $es_{1j_1}, \ldots, es_{mj_m}$, because X needs to satisfy two conditions $|X| = m$ and $h(X) = \{c_1, \ldots, c_m\}$. Now inspect any such event es_{ij}. It can be in C only when:

- if $\alpha_{ij} = x_k$, then also $epx_k \in C$, and
- if $\alpha_{ij} = \neg x_k$, then also $enx_k \in C$.

But these are exactly the cases when $V(x_k)$ is set to *true* and *false*, respectively. (Because the events epx_i and enx_i are in conflict.) Therefore for all clauses $i \in \{1, \ldots, m\}$, there exists a variable x_{k_i}, which satisfies the clause C_i, and therefore the formula ϕ is satisfied by the truth valuation V.

Therefore the prefix β' has a possible extension iff the formula ϕ has a satisfying truth assignment. □

Theorem 10 *POSSIBLE-EXTENSION is NP-complete.*

Proof:
The proof is immediate by Lemmas 8 and 9. □

An example reduction for the 3SAT formula $\phi = ((x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3))$ is presented next. Figure 4.4 represents the net system Σ produced by the reduction (some place labels have been omitted for clarity). Figure 4.5 gives the corresponding prefix β'. The question now is, can β' be extended by the transition t. If this is the case, then ϕ has a satisfying truth assignment. If not, then ϕ is unsatisfiable.

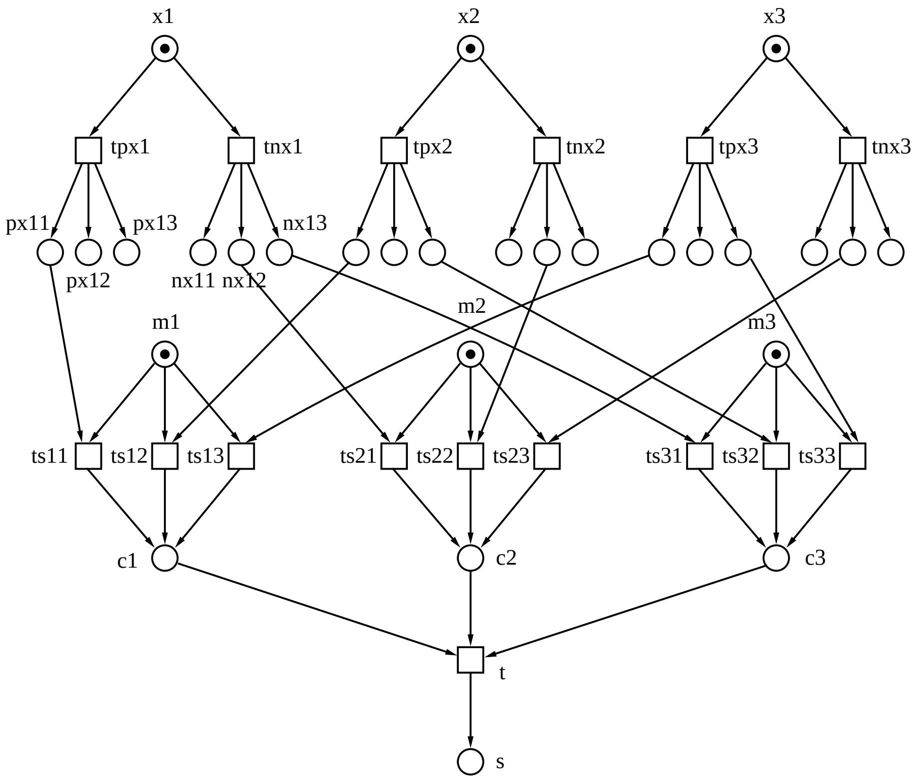


Figure 4.4: Example 3SAT reduction net system Σ.

One can check that, for example, the pair $(t, \{bc_{11}, bc_{23}, bc_{32}\})$ is a possible extension of β', and the corresponding truth valuation is $V(x_1) = true$, $V(x_2) = true$, and $V(x_3) = false$.

As a note we would like to add that the prefix β' will also be generated as a sub-result by the finite complete prefix generation algorithm, Algorithm 2 when using either the adequate order of McMillan [28, 30] or Esparza et.al. [11, 12]. Therefore the NP-hard problem also arises when using a practical prefix generation algorithm.

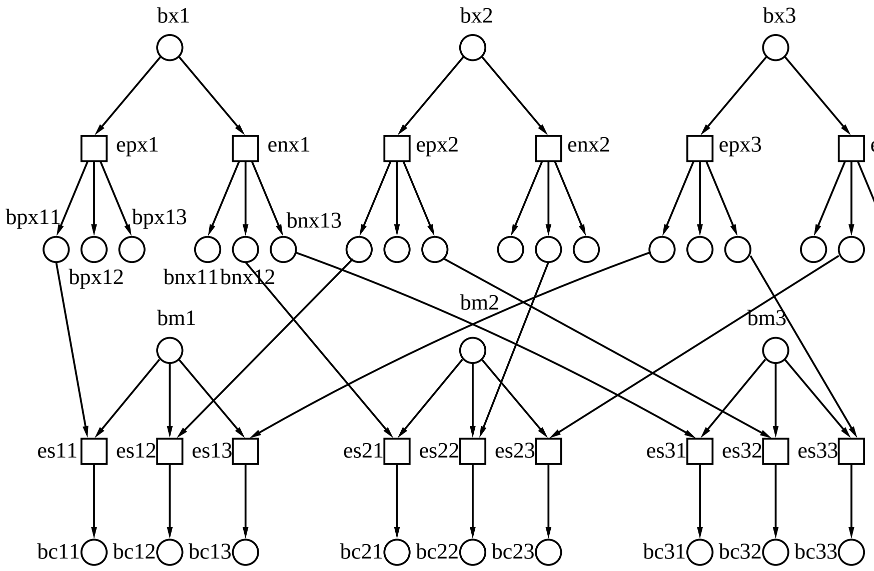


Figure 4.5: Example 3SAT reduction prefix β'.

It is interesting to notice that the NP-hardness proof requires that both the size of the prefix β and the size of the ${}^\bullet t$ (and thus also the size of Σ) need to be increased when reducing a larger 3SAT instance into this problem. In fact, if we put a limit on the in-degree of the transitions of Σ, then there exists a polynomial time algorithm in the size of β to compute POSSIBLE-EXTENSION. This has been noted before, and one way of proving it is to make a co-set checking algorithm similar to Algorithm 3, and to notice that when only the size of the prefix is increased, the number of sets X of the fixed maximum cardinality (or any smaller cardinality) needed to generated for checking, is a polynomial of a fixed degree. Also, if we fix the size of the prefix β, and start increasing the size of the net Σ, the problem stays polynomial in the size of the input. This can be seen from the fact that the guessing part of Algorithm 3 guesses a fixed number of bits, and thus the algorithm can be simulated in polynomial time in the size of the input Σ.

One could argue that the same reduction used in NP-hardness proof could also be done by replacing the transition t with a set of transitions $t_1, \ldots, t_k$ of a fixed in-degree. These transitions would then form a subnet, which "emulates" the effect of transition t using several firings of transitions to emulate one firing of t (atomicity is not a problem, because no transition shares preplaces with t). The resulting prefix would still contain a condition b, such that $h(b) = s$, iff ϕ is satisfiable. However, such a reduction would not prove anything useful about the complexity of the possible extensions subroutine. Each call will in fact have a polynomial running time in the size of the prefix at the time of the call. However, there can now be an exponential number of calls needed to the possible extensions subroutine, and also the size of the "sub-result" prefixes generated can be exponentially larger than the prefix β' used in the proof.

In practice looking at the maximum in-degree of transitions of a net system seems to be the most useful easily obtainable estimate for the hardness of the possible extensions calculation.

5 COMPLEXITY OF MODEL CHECKING

The complexity of model checking with finite complete prefixes is a field in which there are a lot of open questions. McMillan showed that deadlock checking is NP-complete in the size of the finite complete prefix [28, 30]. It is well known that marking reachability and submarking reachability using finite complete prefixes of 1-safe Petri nets are NP-complete in the size of the prefix, however we can not cite a proof in the literature. Therefore we present a proof sketch for this result.

The marking reachability and submarking reachability problem are in NP in the size of the net for acyclic Petri nets. Thus these problems are also in NP in the size of the prefix for finite complete prefixes, see e.g. [9]. To show submarking reachability to be NP-hard in the size of the prefix, we can use the same net Σ' (not containing the transition t) and its prefix β' as in our proof of NP-hardness for the problem POSSIBLE-EXTENSION. Now we ask the question, is there a marking such that all the places $c1, \ldots, c_m$ are marked? It is easy to prove that such a submarking is reachable iff the 3SAT instance ϕ is satisfiable (using a similar proof as in the previous chapter for deciding whether there exists a possible extension by t, as ${}^\bullet t = \{c1, \ldots, c_m\}$).

We need to change the net Σ' to show the same result for marking reachability. The problem is that the places px_{ij} and nx_{ij} might end up containing tokens which cannot be removed. We will fix this problem by adding transitions $tdpx_{ij}$, each of whose preset only contains the corresponding place px_{ij}, and whose postset is empty, and similarly for $tdnx_{ij}$ removing a token from nx_{ij}. (Here we allow transitions with empty postsets for technical convenience, unlike in the rest of this work where all transitions must have non-empty postsets.) It is easy to see that we can still create a polynomial size prefix in polynomial time in the size of the net for this modified net, which contains one event corresponding to each added transition. Now the question asked is: Is the marking $\{c1, \ldots, c_m\}$ reachable? Therefore it is easy to see that the marking reachability problem is also NP-hard.

We can see a 1-safe Petri net system as a representation of its (finite, interleaved) reachability graph. Thus in model checking a Petri net we actually interpret the model checking questions on its reachability graph. Because a finite complete prefix is also a symbolic representation of the reachability graph, we can also do model checking when given a finite complete prefix as input. Next we will introduce the main complexity result to be proved in this chapter. We will prove that given a finite complete prefix of a 1-safe net system Σ, it is PSPACE-complete in the size of the prefix to model check the CTL formula $EF(p \wedge EF(q))$ in the initial state of the net Σ.

Our proof is weak in the sense that we use the semantic definition of finite complete prefixes (Definition 3). There are 1-safe Petri nets for which a minimal finite complete prefix will be polynomial in the size of the original net, however, the prefix generation algorithms presented in the literature (for example Algorithm 2) will create an exponentially larger finite complete prefix. (The only upper bound on the size of the prefix of Algorithm 2 when used with the adequate order of Esparza et.al. is that it is “at most as large” as the reachability graph, which is in our proof exponentially larger than the size of the minimal prefix.) We use one of these nets in the proof, and therefore our result does not directly apply to the prefixes created by for example Algorithm 2. However, we conjecture that our related work on prefix minimization [19] can be modified to generate the needed prefixes. The details of this prefix generation procedure, and its computational complexity are left for further work. Thus our proof should be seen as a proof that semantic definition of finite complete prefixes allows the prefixes to (sometimes) be a very compact symbolic representation of the reachability graph.

Our PSPACE-hardness proof is strong in the sense that the CTL formula $EF(p \wedge EF(q))$ is a fixed length temporal formula, which can be expressed by a lot of different specification languages: UB^-, UB, CTL, CTL*, and deterministic finite automata (see e.g. [5], for UB^- see e.g. [9], and for finite automata see e.g. [26]). Also the proof can be slightly modified to prove the same complexity result for model checking using LTL, Büchi automata (see e.g. [42]), and linear-time μ-calculus (see e.g. [4]) as the specification formalism.

Our proof is based on the proof of PSPACE-hardness of the reachability problem on 1-safe Petri nets. For an introduction, see Esparza's survey paper on complexity results for Petri nets [8]. We use the material and notation of this paper in the proofs.

We define a *Turing machine* to be a tuple $M = \langle Q, \Gamma, \delta, q_0, F\rangle$, where Q is a finite non-empty set of states, Γ a finite non-empty set of tape symbols (containing a special *blank* symbol $\#$), $\delta : (Q \times \Gamma) \mapsto \mathcal{P}(Q \times \Gamma \times \{R, L\})$ is the (non-deterministic) transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We define the *size of a Turing machine* to be the number of bits needed to encode its transition relation. We define a *configuration* of a Turing machine to be a triple $\langle q, w, u\rangle$, where $q \in Q$ is a state, and w, u are strings in Γ^*. The intuition is that q is the current state of the Turing machine, w is the string to the left of the cursor, including the symbol scanned by the cursor, and u is the string to the right of the cursor.

We define the notions of execution and acceptance of a Turing machine in what is in the essence a standard way, see e.g. [36]. However, there are some minor notational differences, which is why we will repeat the definitions here.

We define the relation *yields* to be a relation between two configurations of a Turing machine M, denoted $\langle q, w, u\rangle \stackrel{M}{\rightarrow} \langle q', w', u'\rangle$. It intuitively means that there is a move of the Turing machine M which starts from configuration $\langle q, w, u\rangle$ and ends in the configuration $\langle q', w', u'\rangle$. More formally, it means that for a pair of configurations to be in the yields relation, i.e. $\langle q, w, u\rangle \stackrel{M}{\rightarrow} \langle q', w', u'\rangle$, there must be a move $\langle q', a', D\rangle \in \delta(q, a)$, such that $D \in \{L, R\}$ and either (a) $D = L$, if $|w| \neq 1$ then w' is w with a omitted from its end, and u' is u with a' attached in the beginning, else ($|w| = 1$) $w' = a'$ and $u' = u$; or (b) $D = R$, and w' is w with its last symbol replaced by a', and the first symbol of u appended to it ($\#$ if u is the empty string), and u' is u with the first symbol removed (or if u was the empty string, u' will also be the empty string).

We define an *execution* of a Turing machine $M = \langle Q, \Gamma, \delta, q_0, F\rangle$ to be a sequence of configurations $c_1, c_2, \ldots, c_i$, such that $c_1 \stackrel{M}{\rightarrow} c_2 \stackrel{M}{\rightarrow} \ldots \stackrel{M}{\rightarrow} c_i$, and the first configuration $c_1 = \langle q, w, u \rangle$ is an *initial configuration*, i.e. $q = q_0$. (We will also sometimes have additional constraints on the initial tape state strings w and u. However, these requirements will be specified later in the different applications we present.) We say that an execution is an *accepting execution* if the last configuration $c_i = \langle q_i, w_i, u_i \rangle$ has an accepting state, i.e. $q_i \in F$. We will define that a Turing machine *accepts* if it has an accepting execution starting from some initial configuration (fulfilling the additional application specific constraints put on the initial configurations).

We use slightly nonstandard notation in this chapter. We consider Turing machines which only use a finite tape. They have both a first and a last cell on their tape. As in the standard definition, a move to the left of the first cell results in the machine staying on the first cell. Slightly nonstandard is the handling of the last cell. If the program of the Turing machine tries to move right when being on the last cell, it stays on the last cell. We define a *linearly bounded automaton* to be a Turing machine which uses n tape cells for a Turing machine description of size n. In other words the number of tape cells of the Turing machine with bounded tape is chosen to match the size of the Turing machine. For technical convenience we encode the configuration of a linearly bounded automaton as $\langle q, i, w \rangle$, where q is the control state of the automaton, $i \in \{1, \ldots, n\}$ is the current location of the head of the Turing machine, and $w \in \Gamma^n$ is a string of length n which gives the contents of the n tape cells of the automaton. It is trivial to construct a bijection between the configurations of the standard way of representing configurations, and the one we adapt for linearly bounded automata. Therefore it is just a matter of technical convenience which definition to use.

Many question about the computations of linearly bounded automata are PSPACE-hard. The first one we use is the *empty tape acceptance problem*:

Definition 11 *EMPTY-TAPE ACCEPTANCE:*
Given a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F \rangle$, does A accept on the empty input?

In other words, the problem can be formulated: Is there an accepting execution of A starting from the initial configuration $\langle q_0, 1, \#^n \rangle$? It is well known that EMPTY-TAPE ACCEPTANCE is PSPACE-complete. Moreover, it remains PSPACE-complete if we restrict the automaton A to have only one accepting state q_F, such that $F = \{q_F\}$, see e.g. [8].

We define the size of a 1-safe Petri net system $\Sigma = \langle S, T, F, M_0 \rangle$ to be the number of bits needed to encode the flow relation F, i.e. $\mathcal{O}(|S| \cdot |T|)$. (The 1-safe marking M_0 can be encoded in $\mathcal{O}(|S|)$ space, and thus is also in $\mathcal{O}(|S| \cdot |T|)$.) The main result we use is the following theorem, first proved by Jones, Landweber and Lien [22]:

Theorem 12 *A linearly bounded automaton of size n can be simulated by a 1-safe Petri net of size $\mathcal{O}(n^2)$. Moreover, there is a deterministic polynomial time procedure in the size of the automaton which constructs this net.*

The notion of simulation used above is very strong: a 1-safe Petri net simulates a Turing machine if there is a bijection f between configurations of the machine and markings of the net (which satisfy some additional invariants), such that the machine can move from a configuration c_1 to a configuration c_2 in one step iff the Petri net can move from the marking $f(c_1)$ to the marking $f(c_2)$ by firing exactly one transition.

Let's now introduce the mapping from a linearly bounded automaton to a 1-safe Petri net. Let $A = \langle Q, \Gamma, \delta, q_0, F\rangle$ be a linearly bounded automaton of size n. The computation of A visits at most the cells $c_1, \ldots, c_n$. Let us denote the set of cells with C. The simulating Petri net $N(A)$ contains a place $s(q)$ for each state $q \in Q$, a place $s(c)$ for each cell $c \in C$, and a place $s(a, c)$ for each pair $a \in \Gamma, c \in C$. A token on place $s(q)$ tells that the machine is in state q, a token on $s(c)$ marks the current head location, and when the place $s(a, c)$ is marked it means that the symbol on tape cell c is a. The number of places of the net is $|Q| + n \cdot (1 + |\Gamma|)$.

The transitions of $N(A)$ are obtained from the transition function of A. If $s(q', a', R) \in \delta(q, a)$, then there exists for each cell c a corresponding transition $t(s(q, a, c)w(a')s(q', c'))$, whose input places are $s(q)$, $s(c)$, and $s(a, c)$, and whose output places are $s(q')$, $s(c')$, and $s(a', c)$, where c' is the cell to the right of c, except when c is the last cell, in which case $c' = c$. The left moves are handled similarly, except that now the first cell is an exception, and moving left on it will leave the head on the leftmost cell. The total number of transitions is at most $2 \cdot |Q|^2 \cdot |\Gamma|^2 \cdot n$. Therefore the size of the net is $\mathcal{O}(n^2)$, because the size of A is $\mathcal{O}(|Q|^2 \cdot |\Gamma|^2)$.

The initial 1-safe marking of $N(A)$ puts one token on $s(q_0)$, one on $s(c_1)$, and one on each of the places $s(\#, c_i)$, for all $i \in \{1, \ldots, n\}$. Thus the total size of the net system is $\mathcal{O}(n^2)$.

More formally, we define the mapping from a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F\rangle$ of size n to a 1-safe Petri net $N(A) = \langle S, T, F', M_0\rangle$ as follows (see Figure 5.1 for an example when $|Q| = 3$, $n = 2$, and $\Gamma = \{\#, a, b\}$):

- $S = \{s(q_i) \mid q_i \in Q\} \cup$
  $\{s(c_i) \mid i \in \{1, \ldots, n\}\} \cup$
  $\{s(a, c_i) \mid a \in \Gamma \wedge i \in \{1, \ldots, n\}\}$,

- $T = \{t(s(q, a, c_i)w(a')s(q', c_{i-1})) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge i \in \{2, \ldots, n\} \wedge (q', a', L) \in \delta(q, a)\} \cup$
  $\{t(s(q, a, c_i)w(a')s(q', c_{i+1})) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge i \in \{1, \ldots, n-1\} \wedge (q', a', R) \in \delta(q, a)\} \cup$
  $\{t(s(q, a, c_1)w(a')s(q', c_1)) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge (q', a', L) \in \delta(q, a)\} \cup$
  $\{t(s(q, a, c_n)w(a')s(q', c_n)) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge (q', a', R) \in \delta(q, a)\}$,

- $F' = \{\langle s(q), t(s(q, a, c)w(a')s(q', c'))\rangle$,
  $\langle s(c), t(s(q, a, c)w(a')s(q', c'))\rangle$,
  $\langle s(a, c), t(s(q, a, c)w(a')s(q', c'))\rangle$,
  $\langle t(s(q, a, c)w(a')s(q', c')), s(q')\rangle$,
  $\langle t(s(q, a, c)w(a')s(q', c')), s(c')\rangle$,
  $\langle t(s(q, a, c)w(a')s(q', c')), s(a', c)\rangle \mid$
  $t(s(q, a, c)w(a')s(q', c')) \in T\}$, and

- $M_0 = \{s(q_0), s(c_1)\} \cup$
  $\{s(\#, c_i) \mid i \in \{1, \ldots, n\}\}$.

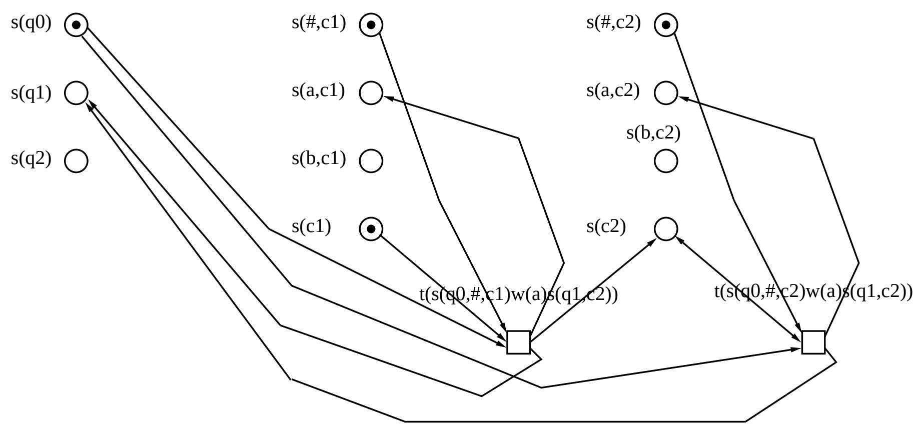


Figure 5.1: A part of $N(A)$ simulating a transition $(q_1, a, R) \in \delta(q_0, \#)$.

The net is also 1-safe, which can be seen by noticing that the initial marking is 1-safe, and then inspecting each transition. The transitions of the net all sustain the following marking invariants:

- $\sum_{q \in Q} M(s(q)) = 1$,
- $\sum_{c \in C} M(s(c)) = 1$, and
- for all $i \in \{1, \ldots, n\} : \sum_{a \in \Gamma} M(s(a, c_i)) = 1$.

Thus the net system $N(A)$ is 1-safe, because the initial marking satisfies the invariants above and the invariants cover all the places of the net.

It follows immediately from this definition that each move of A corresponds to the firing of one transition and vice versa. The configurations reached by A along a computation correspond to the markings reached along its corresponding run. We define the simulation function f in a way which respects the labelling of the places of the net $N(A)$. Namely, the simulation function f maps a configuration $\langle q_i, j, w \rangle$ of the automaton A to a marking M of the net $N(A)$ (satisfying all the invariants above). The marking M satisfies the following constraints (which fully define M): $M(s(q_i)) = 1$, $M(s(c_j)) = 1$, $M(s(a_k, c_k)) = 1$ for all $k \in \{1, \ldots, n\}$ such that $w = a_1 a_2 \ldots a_n$, and $M(s) = 0$ for all other places.

Now given a linearly bounded automaton A, one can in deterministic polynomial time construct the net $N(A)$, and the size of $N(A)$ is polynomial in the size of A. We also assume that the automaton A has only one accepting state q_F. Now to decide the problem EMPTY-TAPE ACCEPTANCE one needs to answer the corresponding problem on the net $N(A)$: Is there a reachable marking of $N(A)$, such that the place $s(q_F)$ is marked?

It is easy to see from the semantics of CTL [5], that the question above is equivalent to the CTL model checking question: $N(A) \models EF(q_F)$? Thus both the reachability of a place, and model checking the CTL formula in question are both PSPACE-hard in the size of the net $N(A)$. For a longer list of PSPACE-hard problems for 1-safe Petri nets, please consult the survey paper [8].

We will next present an alternative proof of PSPACE-hardness of model checking with 1-safe Petri nets. The proof is somewhat more complicated than necessary, if we would only be interested in proving this result for 1-safe Petri nets. However, we want to prove this result also for finite complete prefixes of 1-safe Petri nets. The proofs of the Chapter 5.2 will be based on the construction used in this chapter, and will make that later proof easier. Therefore, we use a slightly different PSPACE-complete problem to make a reduction from:

Definition 13 *ARBITRARY-TAPE-STATE ACCEPTANCE:*
Given a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F \rangle$, does there exists an initial configuration on which A accepts?

In other words: Does there exist an accepting execution of A starting from some initial configuration $\langle q_0, i, w \rangle$, such that $i \in \{1, \ldots, n\}$ and $w \in \Gamma^n$? We have chosen this problem because it allows for an easier proof when applying the proof to model checking with finite complete prefixes of 1-safe Petri nets. In the proofs of this work we use as reductions polynomial-time many-one reductions.

Lemma 14 *ARBITRARY-TAPE-STATE ACCEPTANCE is in PSPACE.*

Proof:
Given a linearly bounded automaton A, generate a nondeterministic (standard) Turing machine $M(A)$ with empty input, and check whether $M(A)$ accepts. Additionally, $M(A)$ never leaves the first $n + 1$ tape cells, and thus by using NPSPACE = PSPACE, see e.g. [36], the language accepted by the machine $M(A)$ is in PSPACE.

The construction used goes as follows. The machine $M(A)$ starts with an empty tape and with the head on the first tape cell. First the machine writes n nondeterministically chosen symbols to the first n cells of the tape. Then the machine writes a new symbol $\triangleleft$, not in the alphabet of A, on the $n + 1$:st cell of the tape to act as an "end-of-tape marker". After this the machine nondeterministically moves the tape head to any cell position $1, \ldots, n$. Finally the machine $M(A)$ starts simulating the machine A, using, for example, two moves to simulate one move of A. The first part of simulating a move changes the state of the machine to simulate the move. The second part checks whether the machine moved on the end marker $\triangleleft$, and if so, moves the head one cell to the left. It is easy to see that $M(A)$ accepts iff A accepts with arbitrary tape state, the size of $M(A)$ is polynomial and it can be constructed in deterministic polynomial time in the size of A, and that $M(A)$ never leaves the first $n + 1$ cells of the tape. Therefore ARBITRARY-TAPE-STATE ACCEPTANCE is in PSPACE. $\square$

Lemma 15 *ARBITRARY-TAPE-STATE ACCEPTANCE is PSPACE-hard.*

Proof:
To show PSPACE-hardness we reduce the EMPTY-TAPE ACCEPTANCE problem to the problem ARBITRARY-TAPE-STATE ACCEPTANCE as follows. Given a linearly bounded automaton A for which we want to solve the EMPTY-TAPE ACCEPTANCE problem, we create a linearly bounded automaton $M(A)$, and solve the ARBITRARY-TAPE-STATE ACCEPTANCE problem.

The constructed machine $M(A)$ first runs a simple fixed-size program, which moves the head of the machine on the first cell of the tape. The machine then writes the blank symbol on the first n cells, where n is the size of A, using a program of size $\mathcal{O}(n)$. Then the machine writes a new symbol $\triangleleft$, not in the alphabet of A, on the $n+1$:st cell of the tape to act as an "end-of-tape marker". The machine then again moves to the first cell of tape using a simple fixed-size program. Finally the machine $M(A)$ starts simulating the machine A, using, for example, two moves to simulate one move of A. The first part of simulating a move changes the state of the machine, and the second part checks whether the machine moved on the end marker $\triangleleft$, and if so, moves the head one cell to the left. It is easy to see that $M(A)$ accepts with arbitrary tape state iff A accepts on empty input. The machine $M(A)$ will also be only polynomially larger than the machine A (and thus has only polynomially more tape cells than A), and can be created in deterministic polynomial time in size of A. This implies ARBITRARY-TAPE-STATE ACCEPTANCE is PSPACE-hard. □

Theorem 16 *ARBITRARY-TAPE-STATE ACCEPTANCE is PSPACE-complete.*

Proof:
The proof is immediate by Lemmas 14 and 15. □

It's also easy to see that also the ARBITRARY-TAPE-STATE ACCEPTANCE problem remains PSPACE-complete even when there is only one accepting state q_F in the automaton A. Given a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F \rangle$ with unique accepting state q_F, we will now reduce the problem ARBITRARY-TAPE-STATE ACCEPTANCE into the model checking problem of the CTL-formula $EF(p \wedge EF(q))$ on a 1-safe net system $\Sigma = C(A)$.

The main intuition behind the reduction is that the mapping $C(A)$ is a "cheating simulation" of the automaton A. The formula takes care of removing all the cheating runs of the net system, and it holds iff the automaton A has an accepting run.

We start proceeding in the same manner as with the reachability checking complexity proof. Our aim is to construct a 1-safe Petri net, which first "nondeterministically" initializes the system into any initial state, and then starts to simulate the behavior of the automaton A starting from this initial state.

The mapping used is very similar to the one used for the reachability proof. The places $s(nq)$, $s(nc)$, and $s(nc_i)$ for all $i \in \{1, \ldots, n\}$ are new. They are used to mark that the control state, tape location, or contents of the tape cell c_i has not been nondeterministically initialized yet. There exists one new transition $t(nq_i)$ for every state of the automaton to initialize the state of the simulation to state q_i. All the transitions $t(nq_i), t(nq_j)$ are pairwise in conflict when $i \neq j$, which ensures that the simulator net is initialized to only one control state. The transitions $t(nc_i)$, and $t(n(a, c_i))$ correspondingly initialize the head location of the automaton, and the contents of each tape cell. The initial marking of the net system is also changed to denote the fact that the nondeterministic initialization needs to be done for state, head location, and each tape cell. Otherwise the mapping is identical to the mapping $N(A)$.

Note that we are even initializing the simulator initial state nondeterministically, instead of initializing it into the initial state q_0. Thus our simulator is a cheating one, and has more executions than a non-cheating one.

We reuse the definition of the net system $N(A) = \langle S, T, F', M_0 \rangle$, and define the net system $\Sigma = C(A) = \langle S', T', F'', M_0' \rangle$ as follows (see Figure 5.2):

- $S' = \{s(nq), s(nc)\} \cup$
  $\{s(nc_i) \mid i \in \{1, \dots, n\}\} \cup$
  S,
- $T' = \{t(nq_i) \mid q_i \in Q\} \cup$
  $\{t(nc_i) \mid i \in \{1, \dots, n\}\} \cup$
  $\{t(n(a, c_i)) \mid a \in \Gamma \wedge i \in \{1, \dots, n\}\} \cup$
  T,
- $F'' = \{\langle s(nq), t(nq_i)\rangle, \langle t(nq_i), s(q_i)\rangle \mid q_i \in Q\}$
  $\{\langle s(nc), t(nc_i)\rangle, \langle t(nc_i), s(c_i)\rangle \mid i \in \{1, \dots, n\}\} \cup$
  $\{\langle s(nc_i), t(n(a, c_i))\rangle, \langle t(n(a, c_i)), s(a, c_i)\rangle \mid a \in \Gamma \wedge i \in \{1, \dots, n\}\} \cup$
  F', and
- $M_0' = \{s(nq), s(nc)\} \cup$
  $\{s(nc_i) \mid i \in \{1, \dots, n\}\}$.

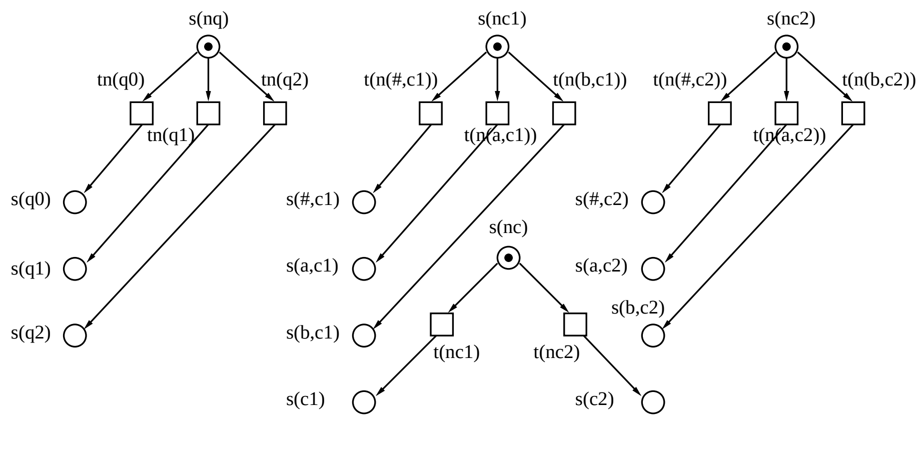


Figure 5.2: The new net structures of $C(A)$, when $|Q| = 3$, $n = 2$, and $\Gamma = \{\#, a, b\}$.

The number of places of the net $C(A)$ is $2 + |Q| + n \cdot (2 + |\Gamma|)$, and the number of transitions is at most $(|\Gamma| + 2 \cdot |Q|^2 \cdot |\Gamma|^2) \cdot n$. Because the initial marking of the net system is 1-safe, the size of the net system $\Sigma = C(A)$ is $\mathcal{O}(n^2)$, where n is the size of the automaton A.

The net is also 1-safe, which can be seen by noticing that the initial marking is 1-safe, and then inspecting each transition. The transitions of the net all sustain the following marking invariants:

- $M(s(nq)) + \sum_{q \in Q} M(s(q)) = 1$,
- $M(s(nc)) + \sum_{c \in C} M(s(c)) = 1$, and
- for all $i \in \{1, \dots, n\} : M(s(nc_i)) + \sum_{a \in \Gamma} M(s(a, c_i)) = 1$.

Thus the net system $C(A)$ is 1-safe, because the initial marking satisfies the invariants above and the invariants cover all the places of the net.

We call a marking which satisfies all the invariants above a *legal marking*. We call a legal marking M *fully initialized* if it also satisfies $M(s(nq)) + M(s(nc)) + \sum_{i \in \{1,\dots,n\}} M(s(nc_i)) = 0$. One can from the invariants above calculate an the number of legal markings (and thus also an upper bound on the number of reachable markings) to be $(|Q| + 1) \cdot (n + 1) \cdot (|\Gamma| + 1)^n$.

Lemma 17 *All legal markings of $C(A)$ are reachable markings.*

Proof:
Inspecting the initial marking of $C(A)$ one can find $n + 2$ sets of enabled transitions. The first set $\{t(nq_0), \dots, t(nq_{|Q|-1})\}$ corresponds to initializing the state, the second set $\{t(nc_1), \dots, t(nq_n)\}$ corresponds to initializing the head location, and the rest n sets $\{t(n(a_1, c_i)), \dots, t(n(a_{|\Gamma|}, c_i))\}$ correspond to setting the contents of the cell i, where $i \in \{1, \dots, n\}$. If we have two transitions t and t' from different sets, then firing t in any marking does not disable t' in the reached marking. It is easy to see that by firing one transition (or by not firing any) from a transition set can put the token of the corresponding invariant to any of its allowed places. Also firing a transition from a set does not touch any of the places mentioned in the other invariants. Therefore, from the initial state of $C(A)$, one can fire sequences of transitions of length $n + 2$ or shorter (one of which is the empty sequence), with which $(|Q| + 1) \cdot (n + 1) \cdot (|\Gamma| + 1)^n$ different markings are reached. □

We will call *initialization transitions* all the transitions of $C(A)$ not already in $N(A)$. The number of fully initialized markings is $|Q| \cdot n \cdot |\Gamma|^n$, and the linearly bounded automaton A has also $|Q| \cdot n \cdot |\Gamma|^n$ configurations. We will now state a correspondence between the fully initialized markings of $C(A)$ and the configurations of A.

Lemma 18 *The exists a simulation bijection f' between the configurations of A and the fully initialized markings of $C(A)$.*

Proof:
Let the net $N(A)$ be initialized to a markings M fulfilling the invariants of $N(A)$, and let $C(A)$ be initialized to a marking M' which agrees on all the places common to both nets and which maps all the places new to $C(A)$ to the empty marking. Let's call this function which maps any marking fulfilling the invariants of $N(A)$ to a fully initialized marking of $C(A)$ as described above the function g, i.e. $g(M) = M'$. We claim that g is a simulation bijection. It is easy to see that whenever $N(A)$ can fire a transition t to reach

the marking M_1, the net $C(A)$ can simulate this move by firing the same transition t to the marking $M_1' = g(M_1)$. Thus $C(A)$ can simulate any move of $N(A)$ and end up in a state related by the function g. To show the simulation in the other direction we just need to notice that in a fully initialized marking M' of $C(A)$ those transitions which only exist in $C(A)$ are not enabled. Therefore, firing any transition t enabled in M' will also be enabled in the marking M of $N(A)$, such that $g(M) = M'$. It's also easy to see that the reached markings M_1' and M_1, respectively, will be related by the same function, i.e. $g(M_1) = M_1'$. Because the cardinalities of fully initialized markings of $C(A)$ and the markings fulfilling the invariants of $N(A)$ agree, the function g is a simulation bijection. Therefore we can create a simulation bijection f' by simple function composition: $f' = g(f(c))$, where c is a configuration of the linearly bounded automaton A. □

Now we will show that the net $C(A)$ can always first fire $n+2$ initialization transitions and after this start simulating A.

Lemma 19 *Let $A = \langle Q, \Gamma, \delta, q_0, F \rangle$ be a linearly bounded automaton, and let $\sigma = c_1, c_2, \ldots, c_m$ be an accepting execution of A, such that c_1 is any initial configuration. Then there exists an execution of the net system $C(A)$: $\sigma' = M_0, M_1, M_2, \ldots, M_{n+2} = f'(c_1), f'(c_2), \ldots, f'(c_n)$, such that M_0 is the initial marking of $C(A)$ and f' is the simulation bijection.*

Proof:
Given the configuration c_1, we construct a sequence of transitions of length $n + 2$. First we add the transition $t(nq_0)$ to initialize the initial state of the simulator to state q_0. Then one needs to find out the head location i of the configuration c_1 and append the transition $t(nc_i)$ to the sequence. Next for all $i \in \{1, \ldots, n\}$: one finds out the symbol a_i on cell i in configuration c_1, and appends the corresponding transition $t(n(a_i, c_i))$ to the sequence. We can use a similar proof as that of Lemma 17 to show that this sequence of transitions is possible to execute in M_0. It is now easy to see from the construction of the transition sequence and the definition of f' that $M_{n+2} = f'(c_1)$, and that M_{n+2} is a fully initialized marking. Thus we can use Lemma 18 to prove by induction that the rest of the execution exists, and will lead to the state $f'(c_n)$. □

In the other direction we are only interested in executions which at some point have the place $s(q_0)$ marked, because all other executions are cheating ones (they never visit the initial state, and will actually be excluded by the formula we model check). First we need to introduce some technical definitions and lemmas to be able to do that.

Definition 20 *A legal marking M is* less initialized *than a legal marking M', if there exists a (possibly empty) sequence of initialization transitions, such that M' is reachable from M by firing this sequence.*

Notice that if M is less initialized than M' and the state has been initialized in M, then the state in M' is the same as the state in M, and correspondingly for the head location and contents of each tape cell. We formalize it as follows:

Lemma 21 *Let M, M' be legal markings of $C(A)$, and s a place both in $C(A)$ and $N(A)$, such that M is less initialized than M' and $M(s) \neq 0$. Then also $M'(s) \neq 0$.*

Proof:
Proof by contradiction. Let $M(s) \neq 0$ and $M'(s) = 0$. Then because M is less initialized than M', there must be a sequence of initialization transitions with which one can reach M' from M. Consider any such sequence σ. Now there must be some initialization transition t in σ, which has s in its preset. However, by static inspection of the net structure of $C(A)$ such initialization transition t does not exist, which leads to a contradiction. □

There is some amount of independence between the initialization transitions and the non-initialization transitions.

Lemma 22 *Let M be a legal marking of $C(A)$, t a non-initialization transition of $C(A)$, and t' an initialization transition of $C(A)$, such that $M \xrightarrow{t} M_1 \xrightarrow{t'} M_1'$. Then there exists a unique legal marking M' of $C(A)$, such that $M \xrightarrow{t'} M' \xrightarrow{t} M_1'$.*

Proof:
We can directly from the structure of the net $C(A)$ see that firing any non-initialization transition t in any marking can never disable an initialization transition t'. Therefore t' must be already enabled at M, and there must exist a marking M' such that $M \xrightarrow{t'} M'$. The marking M' is legal, because its obtained by a firing of a transition with sustains the legality invariant. It is also unique, because the effect of any transition is deterministic in Petri nets. We can also see from the structure of the net that firing any initialization transition t' in any marking will never disable a non-initialization transition t, and thus t' is also enabled at M'. We can now again use the fact that the effects of transitions in Petri nets are deterministic to show that $M' \xrightarrow{t} M_1'$. □

We can express the independence between the initialization transitions and the non-initialization transitions also using the terms of the less initialized relation.

Lemma 23 *Let M be a legal marking of $C(A)$ and t a non-initialization transition of $C(A)$, such that $M \xrightarrow{t} M_1$ and M_1 is less initialized than M_1'. Then there exists a unique legal marking M', such that $M' \xrightarrow{t} M_1'$ and M is less initialized than M'.*

Proof:
Because M_1 is less initialized than M_1', there must exist a (possibly empty) sequence of initialization transitions σ with which M_1' can be reached from M_1. If the sequence is empty $M_1' = M_1$, and thus trivially $M' = M$. Otherwise we prove by induction on the length of the sequence σ using Lemma 22 to prove the induction step (pushing the transition t one step forward into the sequence σ in each induction step), that the transition sequence $\sigma \cdot t$ is

enabled at M and will lead to M'_1. Therefore there must exist a unique legal marking M' such that $M' \xrightarrow{t} M'_1$ and M is less initialized than M', because M can reach M' by the sequence of initialization transitions σ. $\square$

Now we are finally able to state which "accepting executions" of $C(A)$ do not cheat when simulating A.

Lemma 24 *Let q_F be the unique accepting state of a linearly bounded automaton A, and let $\sigma = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots M_{k-1} \xrightarrow{t_k} M_k$ be an execution of $C(A)$, such that there exists $j\ :\ 0 \leq j \leq k \wedge M_j(s(q_0)) \neq 0 \wedge M_k(s(q_F)) \neq 0$, and M_0 is the initial marking of $C(A)$. Then there exists an accepting execution of A which begins from an initial configuration of A.*

Proof:
We do the proof by creating a corresponding execution σ' of A which starts from an initial configuration and ends in an accepting configuration. We do this by induction in the reverse direction of the (suffix) transition sequence $t_{j+1}, \dots, t_k$, i.e. the induction variable i gets the values $k, k-1, \dots, j+1$. In the base case let M'_k be any fully initialized marking such that M_k is less initialized than M'_k (it's easy to see from the structure of $C(A)$ that such M'_k always exists). Let c_k be the unique configuration of A, such that $f'(c_k) = M'_k$, where f' is the simulation bijection, and let the sequence $\sigma'_k = c_k$. In the induction step there are two cases:

- t_i is not an initialization transition: Because $M_{i-1} \xrightarrow{t_i} M_i$, and M_i is less initialized than M'_i, then by Lemma 23 there exists a unique state M'_{i-1}, such that $M'_{i-1} \xrightarrow{t_i} M'_i$. From Lemma 23 we also get that M_{i-1} is less initialized than M'_{i-1}. It is also easy to see that M'_{i-1} is fully initialized, because M'_i is fully initialized, and firing any non-initialization transition t_i in any marking cannot change a marking from non-fully intialized to a fully initialized one. Now let c_{i-1} be the configuration of A, such that $f'(c_{i-1}) = M'_{i-1}$. Then because f' is a simulation bijection, there exists a move of A from c_{i-1} to c_i, and we record that by defining $\sigma'_{i-1} = c_{i-1} \cdot \sigma'_i$.

- t_i is an initialization transition: When $C(A)$ executes an initialization transition, A does not move. This is reflected by defining $M'_{i-1} = M'_i$, $c_{i-1} = c_i$, and $\sigma'_{i-1} = \sigma'_i$. It is now trivial to see that M'_{i-1} is fully initialized. The marking M_{i-1} is also less initialized than M'_{i-1}, because $M_{i-1} \xrightarrow{t_i} M_i$, t_i is an initialization transition, and M_i is less initialized than $M'_i = M'_{i-1}$.

We now claim that the sequence $\sigma' = \sigma'_j$ is an accepting execution of A starting from an initial configuration of A. Because $M_k(s(q_F)) \neq 0$, M_k is less initialized than M'_k and thus by Lemma 21: $M'_k(s(q_F)) \neq 0$, and $f'(c_k) = M'_k$, the configuration c_k is an accepting configuration. Similarly for the marking M_j: Because $M_j(s(q_0)) \neq 0$, M_j is less initialized than M'_j and thus by Lemma 21: $M'_j(s(q_0)) \neq 0$, and $f'(c_j) = M'_j$, the configuration c_j is an initial configuration. One can easily see that a configuration identical to the configuration c_j is the first configuration of the sequence σ', and the configuration c_k is the last configuration. Also it is easy to see that given any two consecutive configurations c_l, c_m of σ', there exists a move of A from c_l to c_m. Therefore σ' is an accepting execution of A starting from an initial configuration of A with arbitrary tape state (namely that specified by c_j). □

Next we make the connection of the property specified above to the temporal logic CTL.

Lemma 25 *For any 1-safe net system N it holds that $N \models EF(p \wedge EF(q))$ iff N has an execution $\sigma = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots M_{k-1} \xrightarrow{t_k} M_k$, such that there exists $j : 0 \leq j \leq k \wedge M_j(p) \neq 0 \wedge M_k(q) \neq 0$, and M_0 is the initial marking of N.*

Proof:
Proof by using the semantic definition of CTL [5], and by simplifying the resulting expression. □

Now we are ready to prove the main theorem about using model checking to solve the ARBITRARY-TAPE-STATE ACCEPTANCE problem:

Theorem 26 *Let A be a linearly bounded automaton with a unique accepting state q_F. It holds that $C(A) \models EF(s(q_0) \wedge EF(s(q_F)))$ iff A has an accepting execution starting from some initial configuration of A.*

Proof:
One direction of the proof is immediate by Lemmas 24 and 25. To prove the other direction we use Lemma 19, and notice that any execution $\sigma' = M_0, \ldots, M_k$ created by its proof starts from the initial state of $C(A)$ and fulfills the property: there exists $j : 0 \leq j \leq k \wedge M_j(s(q_0)) \neq 0 \wedge M_k(s(q_F)) \neq 0$, and thus by using again the Lemma 25 the sequence will be a witness for the formula $EF(s(q_0) \wedge EF(s(q_F)))$ proving the other direction. □

The following lemma is well known, see e.g. [8], but Theorem 26 gives an alternative way of proving it.

Lemma 27 *Model checking a fixed size CTL formula ϕ is PSPACE-hard in the size of the 1-safe net Σ.*

Proof:
One can use the Theorem 26 to reduce the problem ARBITRARY-TAPE-STATE ACCEPTANCE to the problem of CTL model checking a fixed size formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$. The size of the net system $\Sigma = C(A)$ is $\mathcal{O}(n^2)$, i.e. polynomial in the size n of the description of the automaton A, and the reduction can be done in deterministic polynomial time. □

We use the following lemma known from literature:

Lemma 28 *Model checking a fixed size CTL formula ϕ is in PSPACE in the size of the 1-safe net Σ.*

Proof:
See e.g. Esparza's survey paper of Petri net complexity results [8]. □

Theorem 29 *Model checking a fixed size CTL formula ϕ is PSPACE-complete in the size of the 1-safe net Σ.*

Proof:
The proof is immediate by Lemmas 27 and 28. □

We will now make use of the machinery created in the previous section. We will prove model checking complexity results for algorithms which are given a finite complete prefix of a 1-safe Petri net as the input.

We will use the net $C(A)$ as our starting point. We will first define a branching process $\beta_C(A)$, which we will later prove to be a finite complete prefix of $C(A)$. The net structure of $\beta_C(A)$ is identical to the net structure of $C(A)$, if one disregards the postsets of all the non-initialization transitions $t(s(q,a,c)w(a')s(q',c'))$.

An intuition to why the prefix $\beta_C(A)$ is a finite complete prefix of $C(A)$ is the following. All events corresponding to non-initialization transitions will be events whose firing will not reach any new reachable markings. Therefore the prefix generation can be stopped after any such event without loosing any information. This can be seen by the fact that when using steps semantics for Petri nets, all reachable markings of the net system $C(A)$ are reachable in one step using only initialization transitions.

We define a branching process $\beta_C(A) = \langle\langle B, E, F\rangle, h\rangle$ as follows (we this time define events before conditions for notational convenience reasons). See Figure 5.3 for the intialization part of the prefix, together with simulation of one transition of A.

- $E = \{e(nq_i) \mid q_i \in Q\} \cup$
  $\{e(nc_i) \mid i \in \{1, \ldots, n\}\} \cup$
  $\{e(n(a, c_i)) \mid a \in \Gamma \wedge i \in \{1, \ldots, n\}\} \cup$
  $\{e(b(q, a, c_i)w(a')b(q', c_{i-1})) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge i \in \{2, \ldots, n\} \wedge (q', a', L) \in \delta(q, a)\} \cup$
  $\{e(b(q, a, c_i)w(a')b(q', c_{i+1})) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge i \in \{1, \ldots, n-1\} \wedge (q', a', R) \in \delta(q, a)\} \cup$
  $\{e(b(q, a, c_1)w(a')b(q', c_1)) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge (q', a', L) \in \delta(q, a)\} \cup$
  $\{e(b(q, a, c_n)w(a')b(q', c_n)) \mid$
  $q, q' \in Q \wedge a, a' \in \Gamma \wedge (q', a', R) \in \delta(q, a)\}$,

- $B = \{b(nq), b(nc)\} \cup$
  $\{b(nc_i) \mid i \in \{1, \ldots, n\}\} \cup$
  $\{b(q_i) \mid q_i \in Q\} \cup$
  $\{b(c_i) \mid i \in \{1, \ldots, n\}\} \cup$
  $\{b(a, c_i) \mid a \in \Gamma \wedge i \in \{1, \ldots, n\}\} \cup$
  $\{b(q', e(b(q, a, c)w(a')b(q', c'))),$
  $b(c', e(b(q, a, c)w(a')b(q', c'))),$
  $b(a', c, e(b(q, a, c)w(a')b(q', c'))) \mid$
  $e(b(q, a, c)w(a')b(q', c')) \in E\}$,

- $F = \{\langle b(nq), e(nq_i)\rangle, \langle e(nq_i), b(q_i)\rangle \mid q_i \in Q\} \cup$
  $\{\langle b(nc), e(nc_i)\rangle, \langle e(nc_i), b(c_i)\rangle \mid i \in \{1, \ldots, n\}\} \cup$
  $\{\langle b(nc_i), e(n(a, c_i))\rangle, \langle e(n(a, c_i)), b(a, c_i)\rangle \mid a \in \Gamma \wedge i \in \{1, \ldots, n\}\} \cup$
  $\{\langle b(q), e(b(q, a, c)w(a')b(q', c'))\rangle,$
  $\langle b(c), e(b(q, a, c)w(a')b(q', c'))\rangle,$
  $\langle b(a, c), e(b(q, a, c)w(a')b(q', c'))\rangle,$

$\langle e(b(q,a,c)w(a')b(q',c')), b(q', e(b(q,a,c)w(a')b(q',c')))\rangle$,
$\langle e(b(q,a,c)w(a')b(q',c')), b(c', e(b(q,a,c)w(a')b(q',c')))\rangle$,
$\langle e(b(q,a,c)w(a')b(q',c')), b(a', c, e(b(q,a,c)w(a')b(q',c')))\rangle \mid$
$e(b(q,a,c)w(a')b(q',c')) \in E\}$, and

- the mapping h maps each event $e(xyz)$ to the transition $t(xyz)$, and each condition $b(xyz)$ to the place $t(xyz)$, except for the conditions in the postset of the events $e(b(q,a,c)w(a')b(q',c'))$. Namely, each of the conditions $b(q', e(b(q,a,c)w(a')b(q',c')))$ is mapped to $s(q')$, the conditions $b(c', e(b(q,a,c)w(a')b(q',c')))$ to $s(c')$, and the conditions $b(a', c, e(b(q,a,c)w(a')b(q',c')))$ to $s(a', c)$, respectively.

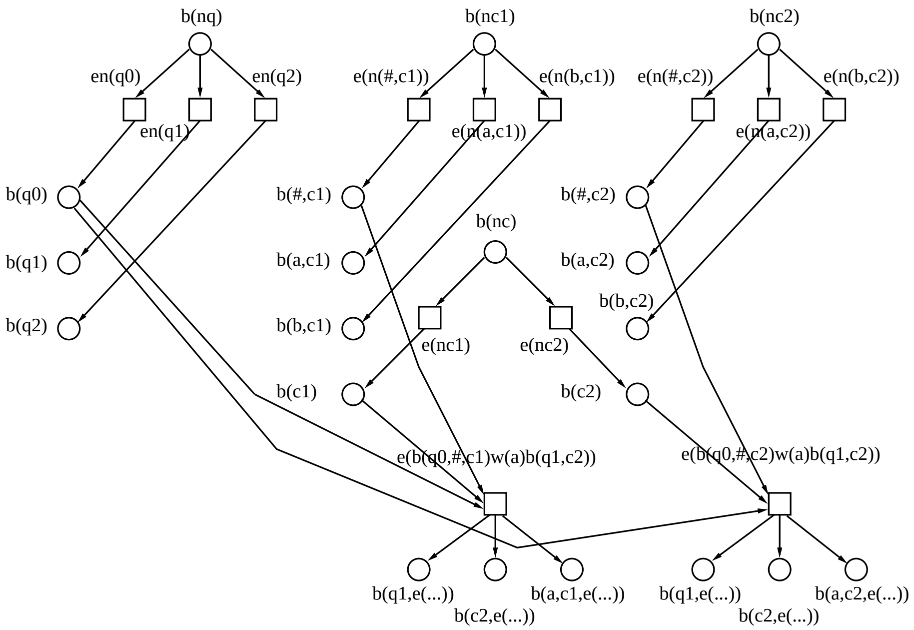


Figure 5.3: A part of $\beta_C(A)$: Initialization and simulation of one transition.

The number of conditions of the prefix $\beta_C(A)$ is $2 + |Q| + n \cdot (2 + |\Gamma| + 6 \cdot |Q|^2 \cdot |\Gamma|^2)$, and the number of events is at most $(|\Gamma| + 2 \cdot |Q|^2 \cdot |\Gamma|^2) \cdot n$. Therefore the size of the branching process $\beta_C(A)$ is polynomial in the size of the net $C(A)$ (and thus also in size of A). It can also be created in deterministic polynomial time in the size of A.

Next we will prove that $\beta_C(A)$ really is a finite complete prefix of $C(A)$.

Lemma 30 *The branching process $\beta_C(A)$ is a finite complete prefix of the unfolding of the net system $C(A)$.*

Proof:
The proof is a quite straightforward but tedious simulation of one possible partial run of the unfolding algorithm, Algorithm 1. First we can see that all the initialization transitions of $C(A)$ are enabled by the initial marking of $C(A)$ and can thus be added as possible extensions to the unfolding. Let's add them all into the unfolding before we continue. It is easy to see that one can create $(|Q|+1)\cdot(n+1)\cdot(|\Gamma|+1)^n$ different configurations using only the events added to the unfolding so far (events corresponding to the initialization

transitions), and that each of those configurations corresponds to a different marking. Because this coincides with the upper bound on the number of reachable markings for $C(A)$, all markings are present in the prefix of the unfolding created so far, and the first criterion of prefix completeness (see Definition 3) is fulfilled.

Consider now the possible extensions of the prefix created so far. None of the initialization transitions can be added as possible extensions, because none of their preplaces appears as a non-minimal condition (all possible extensions containing only minimal conditions were already added to the unfolding). Notice that no place appears so far twice in the unfolding. Let t be any non-initialization transition. The transition t has three places in preset: $s(q), s(c), s(a,c)$. It is easy to see that the set $X = \{b(q), b(c), b(a,c)\}$ is a set such that (t, X) is a possible extension of the unfolding (because the corresponding set of events $\{e(nq), e(nc), e(n(a,c))\}$ is a configuration of the prefix created so far). We can therefore add each non-initialization transition t exactly once to the prefix. When we do that, we notice that we need to create a new condition for each postplace $s(q'), s(c'), s(a',c)$ for each added t. Notice that we again add no conditions corresponding to preplaces of initialization transitions. After we have done this we stop the simulation of the unfolding algorithm.

It is now straighforward to check that the prefix created is isomorphic to $\beta_C(A)$. We also claim that the prefix fulfills the second criterion of completeness (see Definition 3). Let M be any marking of $C(A)$, and let t be a transition enabled at M. If t is an initialization transition we are done, because their preplaces appear only as minimal conditions in the prefix, and all possible extensions of the initial state are already in the prefix. Therefore t is not an initialization transition, and thus has a preset $\{s(q), s(c), s(a,c)\} \subseteq M$. We have proved that we can create a configuration C containing only events corresponding to initialization transitions (and thus added in the first phase) such that $Mark(C) = M$. Let now $Y = Cut(C)$ be the set of conditions corresponding to configuration C. We now know that the set of conditions $X = \{b(q), b(c), b(a,c)\} \subseteq Y$ is a co-set. However, we have in the second phase added the pair (t, X) for all transitions t which are possible extensions. Therefore the prefix contains some event e, such that $C \cup \{e\}$, $e \notin C$, and $h(e) = t$. Thus also the second criterion of prefix completeness is fulfilled.

□

A finite complete prefix contains full information about the reachability graph of the corresponding 1-safe net system (see Definition 3). Thus, CTL model checking can be done by using only the prefix as an input. We first show that model checking using prefixes can also be done in polynomial space by reducing the problem of model checking with prefixes back into the problem of model checking with 1-safe Petri nets. The proof is not very constructive, as it uses the prefix only as an alternative "storage format" for a 1-safe Petri net.

Lemma 31 *Model checking a fixed size CTL formula ϕ is in PSPACE in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
We can by investigating the prefix $\beta = \langle\langle B, E, F\rangle, h\rangle$ generate the net system $\Sigma' = \langle S', T', F', M_0'\rangle$, where:

- $S' = \{s \mid b \in B \,\wedge\, h(b) = s\}$,
- $T' = \{t \mid e \in E \,\wedge\, h(e) = t\}$,
- $F' = \{\langle x', y'\rangle \mid x, y \in (B \cup E) \wedge \langle x, y\rangle \in F \wedge h(x) = x' \wedge h(y) = y'\}$, and
- $M_0' = h(Min(\langle B, E, F\rangle))$.

It is straightforward to prove that the reachability graph of Σ' is isomorphic to the reachability graph of Σ because β a a finite complete prefix of Σ. Also given any reachable marking M of Σ, the corresponding marking M' of Σ' has the same number of tokens on all the places of S', and all other places of M are not marked. Thus the reachability graph of Σ' contains all information about the reachability graph of Σ.

The mapping obviously creates a net system which is at most of the same size as the original prefix. We can also see from definition above that we can implement the mapping in polynomial space in the size of the prefix β. As a technical detail we add the following. If the formula ϕ we want to model check refers to some places not in S', we will add them to S' (i.e. as isolated places with empty initial marking). This can clearly be done in constant time (because the formula size is fixed). Now by Lemma 28 we can run the model checking algorithm for the created net system Σ' in PSPACE in the size of Σ', and get the same result as we would get by model checking the net Σ. The joint complexity of two PSPACE algorithms run after each other is in PSPACE, which completes the proof. □

We can now state the main lemma of this chapter.

Lemma 32 *Model checking a fixed size CTL formula ϕ is PSPACE-hard in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
Theorem 26 gives us a way of reducing the PSPACE-complete problem ARBITRARY-TAPE-STATE ACCEPTANCE (with unique accepting state) to the problem of CTL model checking a fixed size formula ϕ for the 1-safe net system $C(A)$. The size of $C(A)$ is polynomial in the size of A, and it can be created by a deterministic polynomial time algorithm in the size of A.

The size of the finite complete prefix $\beta_C(A)$ of $C(A)$ is polynomial in the size of the net system $C(A)$ (and thus also in the size of A). It is also easy to see that $\beta_C(A)$ can be created by a deterministic polynomial time algorithm which is given $C(A)$ as an input. (The algorithm has to just copy the net structure of $C(A)$ as the net structure of $\beta_C(A)$, with simple constant time special case handling for each non-initialization transition postset. Also creating the function h can be made in polynomial time in the size of $C(A)$ with the same special case handling.) Therefore we can in deterministic polynomial time reduce the problem ARBITRARY-TAPE-STATE ACCEPTANCE (with unique accepting state) to the problem of CTL model checking a fixed size formula ϕ with the finite complete prefix $\beta_C(A)$. Thus the problem is PSPACE-hard in the size of the prefix $\beta_C(A)$. $\square$

We can now sum up the results proved above.

Theorem 33 *Model checking a fixed size CTL formula ϕ is PSPACE-complete in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
The proof is immediate by Lemmas 31 and 32. $\square$

It is easy to see that in general there is no deterministic polynomial time algorithm which generates a finite complete prefix from a given 1-safe net system. However, the net system $C(A)$ is somewhat special. It belongs to a class of 1-safe nets in which all reachable markings can be reached by at most one step in step semantics for Petri nets. For this class of nets the deadlock and reachability problems are in NP in the size of the net system. This is the case because to test for the reachability of any marking, one needs to consider only transition sequences which fire each transition at most once. However, as the proof of Lemma 27 shows, the CTL model checking problem is still PSPACE-hard for this class of nets.

The CTL formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$ syntactically belongs to all the logics UB$^-$, UB, CTL, and CTL*. Therefore the PSPACE-hardness result also applies to them. Any witness path for the formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$ can be seen as a sequence violating a safety property. Therefore is should not come as a surprise that the property can also be expressed by a deterministic finite automaton M with the alphabet $\mathcal{P}(\{q_0, q_F\})$ (see Figure 5.4), leading to a PSPACE-hardness proof for safety model checking.

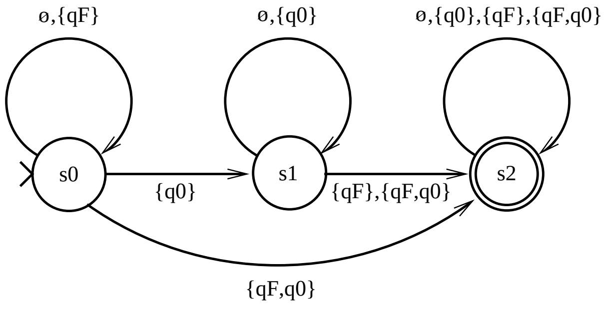


Figure 5.4: The deterministic finite automaton M.

We can also require without loss of generality that all executions of the automaton A entering the final state q_F will keep on looping back to the final state q_F thus creating an infinite execution in which the final state is repeated infinitely often. Now it is easy to see that we can specify a violation of the safety property expressed by the DFA M with a Büchi automaton M'. We have actually defined M in such a way that the automaton M' can be identical to M, only the definition of acceptance is changed to Büchi acceptance. Thus Büchi emptiness checking is also PSPACE-hard. We can also create the LTL formula $\psi = \neg(\Diamond(s(q_0) \wedge \Diamond(s(q_F))))$, which is equivalent to the formula $\Box(\neg(s(q_0)) \vee \Box(\neg(s(q_F))))$ (which one can from the syntax see to be a safety formula as expected). Now it is easy to see that $N \models \phi$ iff $N \not\models \psi$, and therefore LTL-model checking is also PSPACE-hard (because coPSPACE = PSPACE, see e.g. [36]). This LTL formula can also easily be translated into an equivalent linear-time μ-calculus formula, thus obtaining also a PSPACE-hardness result for this logic.

Sometimes we would like to work with marking reachability instead of temporal logics. We know that the reachability problem is NP-complete in the size of a finite complete prefix. To investigate the effects of "nesting of temporal operators", we define a problem similar to our fixed CTL formula model checking, but working with marking reachability.

Definition 34 *REPEATED-REACHABILITY:*
Given two markings M, M' of a 1-safe net system Σ, is the marking M reachable from the initial state of the net system and the marking M' reachable from M.

The complexity result is also in this case similar, but the used reduction differs a bit.

Lemma 35 *The REPEATED-REACHABILITY problem is PSPACE-hard in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
We can reduce the PSPACE-complete problem EMPTY-TAPE ACCEPTANCE with a unique accepting configuration c_F (even in the case that the linearly bounded automaton A has only one accepting configuration, the EMPTY-TAPE ACCEPTANCE problem is still PSPACE-complete, see e.g. [8]) to the problem REPEATED-REACHABILITY. We use again the net system $C(A)$, which is polynomial in the size of A and can be created in deterministic polynomial time from A. As the marking M we use the marking corresponding to the initial configuration with empty tape and head on the first cell, i.e. $M = f'(\langle q_0, 1, \#^n \rangle)$, where f' is the simulation bijection. As the marking M' we use the marking corresponding to the unique accepting configuration $M' = f'(c_F)$.

Because M is a fully initialized marking, it is easy to prove using the simulation bijection f' that the net $C(A)$ simulates the automaton A after reaching the state M. Also it's easy to see that the marking M is always reachable in $C(A)$, and M' is reachable from M iff the automaton A has an accepting execution starting from $\langle q_0, 1, \#^n \rangle$ and ending in the unique accepting configuration c_F.

Therefore the REPEATED-REACHABILITY problem is PSPACE-hard in the size of $C(A)$. Now we can again use the same technique as before and notice that the size of $\beta_C(A)$ is polynomial in the size of $C(A)$, and that it can be constructed by a deterministic polynomial time algorithm from $C(A)$. Therefore we can reduce the problem EMPTY-TAPE ACCEPTANCE (with unique accepting configuration c_F) to the REPEATED-REACHABILITY problem with the finite complete prefix $\beta_C(A)$ in deterministic polynomial time in size of A. Therefore the REPEATED-REACHABILITY problem is

PSPACE-hard in the size of the finite complete prefix of a 1-safe Petri net system. □

Lemma 36 *The REPEATED-REACHABILITY problem is in PSPACE in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
We can use the definition of the net system Σ' as defined in the proof of Lemma 31. The size of the net system Σ' is polynomial in the size of β and can also be created in deterministic polynomial time.

First we check whether the marking M is reachable in Σ'. This can be done in PSPACE, see e.g. [8]. If the marking is not reachable, we reject. Otherwise we construct the net system Σ'', which is identical to Σ' except that the initial marking has been changed to M. Next we check in PSPACE whether M' is reachable in Σ''. If this is the case, we accept otherwise we reject.

It is easy to see that we have correctly solved the problem REPEATED-REACHABILITY, and that the used algorithm is in PSPACE in the size of the prefix β. □

Theorem 37 *The REPEATED-REACHABILITY problem is PSPACE-complete in the size of the finite complete prefix β of a 1-safe net Σ.*

Proof:
The proof is immediate by Lemmas 35 and 36. □

The model checking complexity of all the logics discussed in this work can be proved to be in PSPACE in the size of the finite complete prefix of a 1-safe Petri net system. This can be done by using similar proofs to Lemmas 31 and 36, and known complexity results for model checking 1-safe Petri nets, see e.g. [8].

We can sum up the results on model checking complexity in the size of the finite complete prefix of a 1-safe Petri net as follows:

- Reachability and deadlock checking are NP-complete.
- Model checking a fixed size formula is in PSPACE (for the specification formalisms discussed in this work).
- There are prefixes for which model checking questions which can express some form of "repeated reachability" are PSPACE-hard.
- It is an open question, whether model checking is computationally easier for the prefixes generated by the finite complete prefix generation algorithm, Algorithm 2.

The last point is an interesting one also from an algorithm development point of view: Do the prefixes generated by Algorithm 2 have some special properties which could be exploited by creating a simpler model checking algorithm than that required by arbitrary prefixes?

6 RULE-BASED CONSTRAINT PROGRAMMING

We will use normal logic programs with stable model semantics [14] as the underlying formalism into which the deadlock and reachability problems for finite complete prefixes of 1-safe Petri nets are translated. This chapter is to a large extent based on [37].

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* [34], where stable models are the solutions of the program rules seen as constraints. We consider normal logic programs that consist of rules of the form

$$\mathtt{h} \leftarrow \mathtt{a}_1, \ldots, \mathtt{a}_n, \mathit{not}\,(\mathtt{b}_1), \ldots, \mathit{not}\,(\mathtt{b}_m) \tag{6.1}$$

where $\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{b}_1, \ldots, \mathtt{b}_m$ and $\mathtt{h}$ are propositional atoms. Such a rule can be seen as a constraint saying that if atoms $\mathtt{a}_1, \ldots, \mathtt{a}_n$ are in a model and atoms $\mathtt{b}_1, \ldots, \mathtt{b}_m$ are not in the model, then the atom $\mathtt{h}$ is in the model. 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.

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

P1: $\mathtt{a} \leftarrow \mathit{not}\,(\mathtt{b})$	P2: $\mathtt{a} \leftarrow \mathtt{a}$	P3: $\mathtt{a} \leftarrow \mathit{not}\,(\mathtt{a})$	P4: $\mathtt{a} \leftarrow \mathtt{c}, \mathit{not}\,(\mathtt{b})$
$\mathtt{b} \leftarrow \mathit{not}\,(\mathtt{a})$			$\mathtt{b} \leftarrow \mathit{not}\,(\mathtt{a})$

Program P1 has two stable models: $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$. The property of this program is that we may freely make negative assumptions as long as we do not bump into any contradictions. For example, we may assume $\mathit{not}\,(\mathtt{b})$ in order to deduce the stable model $\{\mathtt{a}\}$. Program P2 has the empty set as its unique stable model. This exposes the fact that we can't use positive assumptions to deduce what is to be included in a model. Program P3 is an example of a program which has no stable models. If we assume $\mathit{not}\,(\mathtt{a})$, then we will deduce $\mathtt{a}$, which will contradict with our assumption $\mathit{not}\,(\mathtt{a})$. Program P4 has one stable model $\{\mathtt{b}\}$. If we assume $\mathit{not}\,(\mathtt{a})$ then we will deduce $\mathtt{b}$. If we assume $\mathit{not}\,(\mathtt{b})$ then we can't deduce $\mathtt{a}$, because $\mathtt{c}$ can't be deduced from our assumptions.

The stable model semantics for a normal logic program P is defined as follows [14]. The reduct P^A of P with respect to the set of atoms A is obtained (i) by 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. A set of atoms A is a stable model of P if and only if A is the deductive closure of P^A when the rules in P^A are seen as inference rules.

A non-deterministic way of constructing stable models is to guess which assumptions (not-atoms of the program) to use, and then check using the deductive closure (in linear time) whether the resulting model agrees with the assumptions. The problem of determining the existence of a stable model is in fact NP-complete [27].

Next we give rest of the stable model semantics definitions, which are needed in the proofs.

Definition 38 *Let A be a set of atoms, we define $not\,(A) = \{not\,(\mathtt{a}) \mid \mathtt{a} \in A\}$.*

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 $Atoms(P)$ 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 $Atoms(P) \setminus \Delta$.

Definition 39 *The deductive closure of a set of rules P and a set of literals B is denoted by $Dcl\,(P, B)$, where $Dcl\,(P, B)$ is the smallest set of atoms that contains B^+ and is closed under $R\,(P, B)$ when*

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

is seen as a set of inference rules.

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

Proposition 40 *The set Δ is a stable model of a set of rules P iff $\Delta = Dcl\,(P, not\,(\overline{\Delta}))$.*

Proof:
Note that the reduct $P^{\Delta} = R\,(P, not\,(\overline{\Delta}))$. □

6.1 THE TOOL SMODELS

There is a tool, the `smodels` system [35, 37], 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 a backtracking search technique similar to the Davis-Putnam method (see e.g. [13]), and it uses a generalization of the well-founded semantics [41] to approximate the stable models and to prune the search space. The backtacking algorithm is based on binary branching on atoms of the program either belonging to a stable model or not. The implementation remembers the number of all such *choice points* which were done during the backtracking search. The `smodels` implementation needs space linear in the size of the input program [37]. 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 including planning and propositional satisfiability checking, see, e.g. [37].

The stable model semantics is defined using rules of the form (1). The `smodels` 2 handles extended rule types [38], which can be seen as succinct encodings of sets of basic rules. One of the rule types is a rule 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 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: $\mathtt{h} \leftarrow \mathtt{a_i}, \mathtt{a_j}$. Using an extended rule instead of the corresponding basic rule encoding was necessary to achieve a linear-size translation of the two problems at hand.

We also use the so called *integrity rules* in the programs. They are rules with no head, i.e. of the form: $\leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, not\,(\mathtt{b_1}), \ldots, not\,(\mathtt{b_m})$. The semantics is given by the following: A new atom $\mathtt{f}$ is introduced to the program, and we will replace the integrity rule with the following rule: $\mathtt{f} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, not\,(\mathtt{b_1}), \ldots, not\,(\mathtt{b_m}), not\,(\mathtt{f})$. 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 doesn't create any new stable models, and neither does adding any set of integrity rules. The last extended rule we use is of the form: $\{\mathtt{h}\} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}$. The semantics is the following: A new atom $\mathtt{h}'$ is introduced to the program, and the rule is replaced by two rules: $\mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, not\,(\mathtt{h}')$, and $\mathtt{h}' \leftarrow not\,(\mathtt{h})$. 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.

7 TRANSLATING DEADLOCK AND REACHABILITY CHECKING INTO LOGIC PROGRAMS

Petri nets are a widely used model for analyzing concurrent and distributed systems. Often such a system must exhibit reactive, non-terminating behavior, and one of the key analysis problems is that of deadlock-freedom: Do all reachable global states of the system (markings of the net) enable some action (net transition)? In this work we study this problem for a subclass of Petri nets, the 1-safe Petri nets, which are capable of modeling finite-state systems. For 1-safe Petri nets the deadlock detection problem is PSPACE-complete in the size of the net [9], however, restricted subclasses of 1-safe Petri nets exist for which this problem is NP-complete [28, 30]. McMillan has presented a deadlock detection method for Petri nets based on finite complete prefixes (i.e. net unfoldings) [28, 30]. The basic idea is to transform the PSPACE-complete deadlock detection problem for a 1-safe Petri net into a potentially exponentially larger NP-complete problem. This translation creates a finite complete prefix, which is an acyclic 1-safe Petri net of a restricted form. Experimental results show that the blowup of the transformation can in many cases be avoided [11, 12, 28, 30, 32].

In this work we address the NP-complete deadlock detection problem for finite complete prefixes. McMillan originally suggested a branch-and-bound algorithm for solving this problem. Recently, Melzer and Römer have presented another algorithm which is based on solving mixed integer programming problems generated from prefixes [32]. Their approach seems to be faster than McMillan's on examples in which a large percentage of the events of the prefix are so called cut-off events. However, if this assumption does not hold, the run times are generally slower than those of the McMillan's algorithm [32].

In this work we study an approach that is similar to that of Melzer and Römer in the way of being capable of handling cases with a large percentage of cut-off events but with more competitive performance. Instead of mixed integer programming our approach is based on a constraint-based logic programming framework [34, 35, 37]. We translate the deadlock detection problem into the problem of finding a stable model of a logic program. We also devise such a translation for checking the reachability problem, which is also NP-complete in the size of the prefix, see a proof sketch in the beginning of Chapter 5. The main contribution of this Chapter also includes the detailed correctness proofs of the translations. For the deadlock detection problem we present benchmark results, and find our approach competitive with the two previous finite complete prefix based approaches.

In this chapter we present the translations of deadlock and reachability properties into logic programs with stable model semantics. The main part of the work presented here has been published in [18, 20, 21], but this work contains some new experimental results. For the deadlock property the main result can be seen as a rephrasing of the Theorem 4 of [32], where mixed integer programming has been replaced by the rule-based constraint programming framework. For the reachability property we give another translation. In this work we assume that the set of events of a finite complete prefix is non-empty. If it is empty, the corresponding net system would have no events enabled in the initial state, and then the deadlock and reachability properties can be trivially solved by looking at the initial state only.

Correctness proofs of the translations are done in two steps. First a program is constructed whose stable models are proved to have a one-to-one correspondence with the configurations of the finite complete prefix which contain no cut-off events. Then additional rules added by the translations are shown to either remove all potential stable models corresponding to live configurations, or all potential stable models which do not agree with the used assertion, depending on the translation.

7.1 BASIC TRANSLATION

Now we are ready to define our translation from the finite complete prefixes into logic programs with stable model semantics. The basic part of our translation is given next. It translates the notion of a configuration of a finite complete prefix into the problem of finding a stable model of a logic program. The definitions will be followed by an example translation given in Fig. 7.1.

First we define some additional notation. We assume a unique numbering of the events (and conditions) of the finite complete prefix. We use the notation e_i (b_i) to refer to the event (condition) number i. In the logic programs $\mathtt{e_i}$, ($\mathtt{b_i}$) is an atom of the logic program corresponding to the event e_i (condition b_i). In the logic program definitions we use the convention that a part of a rule will be omitted, if the corresponding set evaluates to the empty set. For example, rule of type 1 of Definition 41 below for an event e_i, such that ${}^\bullet({}^\bullet e_i) = \emptyset$, would become: $\mathtt{e_i} \leftarrow \mathit{not}\,(\mathtt{be_i})$.

Definition 41 *Let $\beta = \langle N, h \rangle$ with $N = \langle B, E, F \rangle$ be a finite complete prefix of a given 1-safe net system Σ. Let $P_B(\beta)$ be a logic program containing the following rules:*

1. *For all $e_i \in E \setminus \mathit{CutOffs}(\beta)$ a rule:*
   $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}, \mathit{not}\,(\mathtt{be_i})$,
   such that $\{e_{p_1}, \ldots, e_{p_n}\} = {}^\bullet({}^\bullet e_i)$.

2. *For all $e_i \in E \setminus \mathit{CutOffs}(\beta)$ a rule:*
   $\mathtt{be_i} \leftarrow \mathit{not}\,(\mathtt{e_i})$.

3. *For all $b_i \in B$ such that $|b_i{}^\bullet \setminus \mathit{CutOffs}(\beta)| \geq 2$ a rule:*
   $\leftarrow 2\{\mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}\}$,
   such that $\{e_{p_1}, \ldots, e_{p_n}\} = b_i{}^\bullet \setminus \mathit{CutOffs}(\beta)$.

The intuition behind the rules of the program $P_B(\beta)$ are the following. Rules of the type 1 give the preconditions under which an atom corresponding to each event can exist in a configuration. Rules of the type 2 enable an event not to be included in a configuration even if its preconditions are present.

Rules of the type 3 disallow all sets of events which contain events in a conflict. Note that because in prefixes each condition has only one event in its preset, the program above does not need atoms corresponding to the conditions of the prefix. The translation above could be trivially extended to also include the cut-off events, but they are not needed by the applications in this work.

We define a mapping from a set of events of the prefix to a set of atoms of a logic program and vice versa.

Definition 42 *The set of atoms of a logic program P corresponding to a set of events $C \subseteq E \setminus Cutoffs(\beta)$ of a finite complete prefix β is $Model(C) = \{\mathtt{e_i} \mid e_i \in C\} \cup \{\mathtt{be_j} \mid e_j \in E \setminus \{C \cup Cutoffs(\beta)\}\}$.*

Definition 43 *The set of events corresponding to a stable model Δ of a logic program P is $Events(\Delta) = \{e_i \in E \mid \mathtt{e_i} \in \Delta\}$.*

Now we are ready to state the correspondence between the finite complete prefix and the basic part of our translation.

Theorem 44 *Let β be a finite complete prefix of a 1-safe net system Σ, let $P_B(\beta)$ be the logic program translation by Definition 41, and let C be a configuration of β, such that $C \cap Cutoffs(\beta) = \emptyset$. Then the set of atoms $\Delta = Model(C)$ is a stable model of $P_B(\beta)$. Additionally, the mapping $Events(\Delta)$ is a bijective mapping from the stable models of $P_B(\beta)$ to the configurations of β which contain no cut-off events.*

We prove the correspondence between the stable models of the program $P_B(\beta)$ and the configurations of β which contain no cut-off events. We proceed by first proving auxiliary Lemma 45. In Lemmas 47 and 48 one direction of the correspondence between stable models of the program $P_B(\beta)$ and the configurations of β with no cut-off events is proved. The Lemma 49 shows the other direction of the correspondence, and together with Corollary 46 shows that the mapping $Events(\Delta)$ is a bijective mapping between the two sets in question.

Lemma 45 *Let Δ be a stable model of program $P_B(\beta)$ obtained from the translation by Definition 41, and $\mathtt{e_i}, \mathtt{be_i} \in Atoms(P_B(\beta))$. Now the following holds: $\mathtt{e_i} \in \Delta \Rightarrow \mathtt{be_i} \in \overline{\Delta}$, and $\mathtt{e_i} \in \overline{\Delta} \Rightarrow \mathtt{be_i} \in \Delta$.*

Proof:
Assume that $\mathtt{e_i} \in \Delta$. The only rule in which $\mathtt{be_i}$ appears as a head is a rule of type 2 of the form $\mathtt{be_i} \leftarrow not\,(\mathtt{e_i})$. By the definition of stable models, because $\mathtt{e_i} \in \Delta$, the reduct $R\,(P_B(\beta), not(\overline{\Delta}))$ does not contain this rule, and thus $\mathtt{be_i} \notin Dcl(P_B(\beta), not(\overline{\Delta}))$, which implies $\mathtt{be_i} \in \overline{\Delta}$, because Δ is a stable model.

Assume that $\mathtt{e_i} \in \overline{\Delta}$. Therefore $R\,(P_B(\beta), not(\overline{\Delta}))$ contains the rule $\mathtt{be_i} \leftarrow$, and thus $\mathtt{be_i} \in Dcl(P_B(\beta), not(\overline{\Delta}))$, which implies $\mathtt{be_i} \in \Delta$, because Δ is a stable model. □

Corollary 46 *Let Δ be any stable model of $P_B(\beta)$. The set of events $C = Events(\Delta)$ fully specifies Δ, i.e. the following holds: $Model(Events(\Delta)) = \Delta$.*

Now we do the main proof of Theorem 44. As the first step we use a subset of the rules of the program $P_B(\beta)$ and prove that for this subset of rules the set of atoms $\Delta = Model(C)$ is a stable model.

Lemma 47 *Let C be a configuration of β such that $C \cap Cutoffs(\beta) = \emptyset$, and let $P_A(\beta)$ be the logic program containing only the rules of the types 1 and 2 of the program $P_B(\beta)$. The set of atoms $\Delta = Model(C)$ is a stable model of $P_A(\beta)$.*

Proof:
We need to prove that $Dcl(P_A(\beta), not(\overline{\Delta})) = \Delta$. We proceed by case analysis on the pairs of atoms $\{\mathtt{e_i}, \mathtt{be_i}\}$ in $Atoms(P_A(\beta))$ showing for both atoms that: For each atom $a \in \Delta$ that $a \in Dcl(P_A(\beta), not(\overline{\Delta}))$, and for each atom $a \in \overline{\Delta}$ that $a \notin Dcl(P_A(\beta), not(\overline{\Delta}))$.

1. Investigate the pair of atoms $\{\mathtt{e_i}, \mathtt{be_i}\}$ such that $\mathtt{e_i} \in \overline{\Delta}$, and thus by Definition 42: $\mathtt{be_i} \in \Delta$. The reduct $R\left(P_A(\beta), not(\overline{\Delta})\right)$ contains a rule of the form $\mathtt{be_i} \leftarrow$, and also the only way of deducing $\mathtt{e_i}$, the rule of type 1 of the form: $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, is not in the reduct, which implies $\mathtt{be_i} \in Dcl(P_A(\beta), not(\overline{\Delta}))$ and $\mathtt{e_i} \notin Dcl(P_A(\beta), not(\overline{\Delta}))$.

2. Investigate the pair of atoms $\{\mathtt{e_i}, \mathtt{be_i}\}$ such that $\mathtt{e_i} \in \Delta$, and thus by Definition 42: $\mathtt{be_i} \in \overline{\Delta}$. The reduct $R\left(P_A(\beta), not(\overline{\Delta})\right)$ does not contain the rule of type 2 of the form: $\mathtt{be_i} \leftarrow$, which implies $\mathtt{be_i} \notin Dcl(P_A(\beta), not(\overline{\Delta}))$. The reduct contains a rule of type 1 of the form: $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, such that $\{e_{p_1}, \ldots, e_{p_n}\} = {}^\bullet({}^\bullet e_i)$. We need an induction to complete the proof for this case.

What is left to be proved is that $\mathtt{e_j} \in Dcl(P_A(\beta), not(\overline{\Delta}))$ for all $\mathtt{e_j} \in \Delta$, and thus for all $e_j \in C$. If $C = \emptyset$ we are done. Otherwise, we prove the previous claim by induction on the index number k of a sequence of events: $e'_1, \ldots, e'_{|C|}$, such that $\{e'_1, \ldots, e'_{|C|}\} = C$, and for all $k \in \{1, \ldots, |C|\}$ it holds that ${}^\bullet({}^\bullet e'_k) \subseteq \bigcup_{1 \le l < k}\{e'_l\}$. Because C is a configuration, such a sequence (a causal total order of events of a configuration) must exist. (This is the case because each non-empty configuration corresponds to a set of execution sequences, and we can choose any sequence from this set.) We pick one such a sequence.

- *Base case $k = 1$:* For e'_1 it holds that ${}^\bullet({}^\bullet e'_1) = \emptyset$. Thus the reduct has a rule of the form $\mathtt{e'_1} \leftarrow$, which implies $\mathtt{e'_1} \in Dcl(P_A(\beta), not(\overline{\Delta}))$.

- *Inductive case $k > 1$:* For all $1 \le j < k$ the induction claim $\mathtt{e'_j} \in Dcl(P_A(\beta), not(\overline{\Delta}))$ holds by the inductive hypothesis. Because ${}^\bullet({}^\bullet e'_k) \subseteq \bigcup_{1 \le l < k}\{e'_l\}$, the reduct has a rule $\mathtt{e'_k} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, such that $\{\mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}\} \subseteq Dcl(P_A(\beta), not(\overline{\Delta}))$ by the inductive hypothesis, which implies $\mathtt{e'_k} \in Dcl(P_A(\beta), not(\overline{\Delta}))$.

The union of all the pairs equals $Atoms(P_A(\beta))$, which implies Δ is a stable model of $P_A(\beta)$. □

Now we continue our proof by considering the full program $P_B(\beta)$ in Lemma 48.

Lemma 48 *Let C be a configuration of β such that $C \cap \mathit{Cutoffs}(\beta) = \emptyset$. The set of atoms $\Delta = \mathit{Model}(C)$ is a stable model of the program $P_B(\beta)$.*

Proof:
By Lemma 47: $\Delta = \mathit{Model}(C)$ is a stable model of $P_A(\beta)$. In the program $P_B(\beta)$ only integrity rules of type 3 of Definition 41 have been added. Thus the set of stable models of $P_B(\beta)$ is always a subset of the stable models of $P_A(\beta)$. Because C is a configuration, there does not exists two non-cut-off events e_i and e_j, and a condition b_k such that: $e_i \neq e_j$, and $b_k \in {}^\bullet e_i \cap {}^\bullet e_j$. Therefore there is no integrity rule which could be used, which implies Δ is a stable model of $P_B(\beta)$. □

Lemma 49 *If Δ is a stable model of the program $P_B(\beta)$, then the set of events $C = \mathit{Events}(\Delta)$ is a configuration of β such that $C \cap \mathit{Cutoffs}(\beta) = \emptyset$.*

Proof:
Because Corollary 46 says that any stable model is fully specified by a set of events, to get a contradiction we need to find a set of events E', which is not a configuration, $E' \cap \mathit{Cutoffs}(\beta) = \emptyset$, and $\Delta = \mathit{Model}(E')$ is a stable model of $P_B(\beta)$. There are two cases:

1. Assume that E' is not causally closed. Thus there must exist an event $e_i \in E'$, such that ${}^\bullet({}^\bullet e_i) \setminus E' \neq \emptyset$. Now by Definition 42 and Lemma 45: $\mathtt{be_i} \notin \mathit{Model}(E')$, and thus the reduct $R\left(P_B(\beta), \mathit{not}(\overline{\Delta})\right)$ contains a rule of the form $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, which is the only rule in which $\mathtt{e_i}$ appears as a head. However, by ${}^\bullet({}^\bullet e_i) \setminus E' \neq \emptyset$ and Definition 42: $\mathtt{be_j} \in \mathit{Model}(E')$ for some $\mathtt{j} \in \{\mathtt{p_1}, \ldots, \mathtt{p_n}\}$. Thus by Lemma 45: $\mathtt{e_j} \notin \Delta$, and thus $\mathtt{e_i} \notin \mathit{Dcl}(P_B(\beta), \mathit{not}(\overline{\Delta}))$, which implies Δ is not a stable model.

2. Assume that E' contains a conflict. Thus there exists two non-cut-off events e_i and e_j, and a condition b_k such that: $e_i \neq e_j$, and $b_k \in {}^\bullet e_i \cap {}^\bullet e_j$. The integrity rule of type 3 of Definition 41 corresponding to the condition b_k has the form $\leftarrow 2\{\mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}\}$, such that $\{\mathtt{e_i}, \mathtt{e_j}\} \subseteq \Delta$, which eliminates the possibility that Δ is a stable model.

□

Proof of Theorem 44:
The Lemmas 48 and 49 prove the correspondence between the configurations of the prefix β which contain no cut-off events, and the stable models of the program $P_B(\beta)$. Combined with Corollary 46 the bijectivity between these two sets is shown. This completes the proof. □

7.2 DEADLOCK TRANSLATION

Next we move to the deadlock translation. We add a set of rules to the program which place additional constraints on the stable models of the program $P_B(\beta)$. We add integrity rules to the program, which remove all stable models of the basic program which are not deadlocks. To do this we model the the enabling of each event (cut-off or not) of the prefix in the logic program.

Definition 50 *Let $\beta = \langle\langle B, E, F\rangle, h\rangle$ be a finite complete prefix of a given 1-safe net system Σ. Let $P_D(\beta)$ be a logic program containing all the rules of the program $P_B(\beta)$ of Definition 41, and also the following rules:*

1. *For all $b_i \in \{b_j \in B \mid {b_j}^{\bullet} \neq \emptyset\}$ a rule:*
   $\mathtt{b_i} \leftarrow \mathtt{e_l}, \mathit{not}\,(\mathtt{e_{p_1}}), \ldots, \mathit{not}\,(\mathtt{e_{p_n}})$,
   such that $\{e_l\} = {}^{\bullet}b_i$, and $\{e_{p_1}, \ldots, e_{p_n}\} = {b_i}^{\bullet} \setminus \mathit{CutOffs}(\beta)$.

2. *For all $e_i \in E$ a rule:*
   $\leftarrow \mathtt{b_{p_1}}, \ldots, \mathtt{b_{p_n}}$,
   such that $\{b_{p_1}, \ldots, b_{p_n}\} = {}^{\bullet}e_i$.

Theorem 51 *Let β be a finite complete prefix of a 1-safe net system Σ, and let $P_D(\beta)$ be the logic program translation by Definition 50. There exists a stable model of $P_D(\beta)$ iff Σ has a reachable deadlock marking. Additionally, for any stable model Δ of $P_D(\beta)$, the set of events $C = \mathit{Events}(\Delta)$ is a deadlock configuration of β, such that $\mathit{Mark}(C)$ is a reachable deadlock marking of Σ.*

First we give Lemma 52, which enables us to add rules of a restricted form into a logic program.

Lemma 52 *Let P_1, P_2 be a logic programs such that for each rule in P_2 the head of the rule is not in $\mathit{Atoms}(P_1)$, and all its body literals are in $\mathit{Atoms}(P_1)$. Then:*

- *If Δ is a stable model of P_1, then $\Delta' = \Delta \cup S(\Delta)$ is the unique stable model of $P_1 \cup P_2$, such that for all $\mathtt{a} \in \mathit{Atoms}(P_1) : \mathtt{a} \in \Delta'$ iff $\mathtt{a} \in \Delta$, where*

$$S(\Delta) = \{\mathtt{h} \mid \mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathit{not}\,(\mathtt{b_1}), \ldots, \mathit{not}\,(\mathtt{b_m}) \in P_2,\ \text{such that}\ \mathtt{a_i} \in \Delta\ \text{for}\ \mathtt{i} = 1, \ldots, \mathtt{n}\ \text{and}\ \mathtt{b_j} \notin \Delta\ \text{for}\ \mathtt{j} = 1, \ldots, \mathtt{m}\}.$$

- *$P_1 \cup P_2$ has the same number of stable models as P_1.*

Proof:
Let $r = \mathtt{h} \leftarrow \mathtt{a_1}, \ldots, \mathtt{a_n}, \mathit{not}\,(\mathtt{b_1}), \ldots, \mathit{not}\,(\mathtt{b_m})$ be any rule of the program P_2, and Δ a stable model of P_1. If we create a program $P_1' = P_1 \cup \{r\}$, then clearly P_1' has the stable model $\Delta' = \Delta \cup \{h\}$ iff $\mathtt{a_i} \in \Delta$ for $\mathtt{i} = 1, \ldots, \mathtt{n}$ and $\mathtt{b_j} \notin \Delta$ for $\mathtt{j} = 1, \ldots, \mathtt{m}$, and $\Delta' = \Delta$ otherwise. Also the possible addition of the atom h into the program P_1' does not effect the reduct i.e. $R\,(P_1', \mathit{not}(\overline{\Delta'})) = R\,(P_1', \mathit{not}(\overline{\Delta}))$, because h doesn't appear as a body literal in any rule in $P_1 \cup P_2$. Therefore the number of stable models remains the same after the addition of r. The claim can be now proved by induction on the number of rules added from P_2. ☐

Corollary 53 *Let P_1 and P_2 be two programs satisfying the requirements of Lemma 52. If Δ is a stable model of $P_1 \cup P_2$, then $\Delta' = \{a \in \Delta \mid \mathit{Atoms}(P_1)\}$ is a stable model of P_1.*

Next we start using Lemma 52 to incrementally prove out translation correct.

Lemma 54 *Let program $P_C(\beta)$ be a program made by adding rules of the type 1 of Definition 50 to the program $P_B(\beta)$. The program $P_C(\beta)$ has the same number of stable models as $P_B(\beta)$, and the stable models agree on the set of atoms $Atoms(P_B(\beta))$.*

Proof:
The proof is immediate by Lemma 52. □

Lemma 55 *For any stable model Δ of $P_C(\beta)$ and any $\mathtt{b_i} \in Atoms(P_C(\beta))$ it holds that $\mathtt{b_i} \in \Delta$ iff $b_i \in Cut(Events(\Delta))$.*

Proof:
Fix any stable model Δ of $P_C(\beta)$, and any atom $\mathtt{b_i}$ of the program. Now the rule corresponding to this atom is: $\mathtt{b_i} \leftarrow \mathtt{e_l}$, *not* $(\mathtt{e_{p_1}}), \ldots,$ *not* $(\mathtt{e_{p_n}})$, such that $\{e_l\} = {}^\bullet b_i$, and $\{e_{p_1}, \ldots, e_{p_n}\} = b_i^\bullet \setminus CutOffs(\beta)$. The corresponding rule will be in the reduct (and by Lemma 52: $\mathtt{b_i} \in \Delta$) iff $\mathtt{e_l} \in \Delta$ and $\mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}} \in \overline{\Delta}$, and thus by Theorem 44 $e_l \in Events(\Delta)$ and $e_{p_1}, \ldots, e_{p_n} \notin Events(\Delta)$, which is exactly the case when $b_i \in Cut(Events(\Delta))$. □

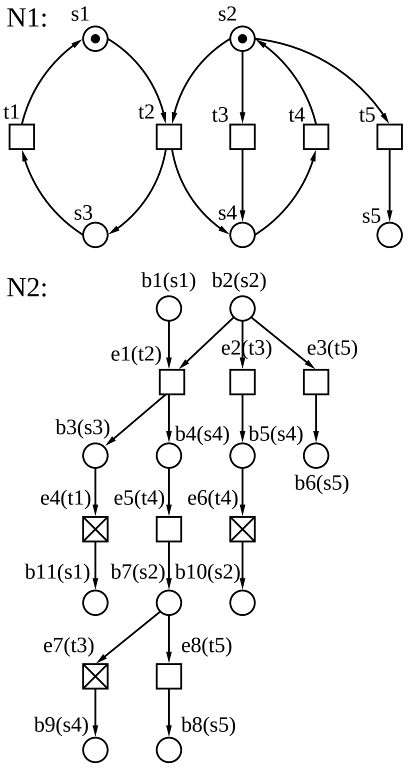
Lemma 56 *Let $P_D(\beta)$ be the logic program translation of the prefix β by Definition 50, and let Δ be a stable model of $P_D(\beta)$. Then the set of events $Events(\Delta)$ is a deadlock configuration of β. Additionally, $P_D(\beta)$ has the same number of stable models as there are deadlock configurations of β, which contain no cut-off events.*

Proof:
The program $P_D(\beta)$ is the program $P_C(\beta)$ with only integrity rules of the type 2 of Definition 50 added. Thus the set of stable models of $P_D(\beta)$ is a subset of the stable models of the program $P_C(\beta)$, which by Lemma 54 correspond to the configurations of β which contain no cut-off events. Fix any stable model Δ of $P_C(\beta)$. There are now two cases to consider:

- $Events(\Delta)$ is not a deadlock configuration of β: Thus there must exist an event e_i (cut-off or not) which is enabled by $Cut(Events(\Delta))$. Consider now the rule of type 2 corresponding to the event e_i of the form: $\leftarrow \mathtt{b_{p_1}}, \ldots, \mathtt{b_{p_n}}$, such that $\{b_{p_1}, \ldots, b_{p_n}\} = {}^\bullet e_i$. Now by Lemma 55 each of the atoms $\mathtt{b_{p_j}} \in \Delta$ iff $b_{p_j} \in Cut(Events(\Delta))$. Thus the integrity rule for the event e_i will be used, which implies Δ is not a stable model of $P_D(\beta)$.

- $Events(\Delta)$ is a deadlock configuration of β: Thus there is no event e_i (cut-off or not) which is enabled by $Cut(Events(\Delta))$. Now by Lemma 55 each of the atoms $\mathtt{b_i} \in \Delta$ iff $b_i \in Cut(Events(\Delta))$. Therefore none of the rules of the type 2 of Definition 50 can be used, which implies Δ is a stable model of $P_D(\beta)$.

We have now found a one-to-one correspondence between the the stable models of $P_D(\beta)$ and the deadlock configurations of β which contain no cut-off events. □



$P_D(N2)$:

$\mathtt{e}_1 \leftarrow \mathit{not}\,(\mathtt{be}_1)$
$\mathtt{be}_1 \leftarrow \mathit{not}\,(\mathtt{e}_1)$
$\mathtt{e}_2 \leftarrow \mathit{not}\,(\mathtt{be}_2)$
$\mathtt{be}_2 \leftarrow \mathit{not}\,(\mathtt{e}_2)$
$\mathtt{e}_3 \leftarrow \mathit{not}\,(\mathtt{be}_3)$
$\mathtt{be}_3 \leftarrow \mathit{not}\,(\mathtt{e}_3)$
$\mathtt{e}_5 \leftarrow \mathtt{e}_1, \mathit{not}\,(\mathtt{be}_5)$
$\mathtt{be}_5 \leftarrow \mathit{not}\,(\mathtt{e}_5)$
$\mathtt{e}_8 \leftarrow \mathtt{e}_5, \mathit{not}\,(\mathtt{be}_8)$
$\mathtt{be}_8 \leftarrow \mathit{not}\,(\mathtt{e}_8)$
$\leftarrow 2\{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3\}$
$\mathtt{b}_1 \leftarrow \mathit{not}\,(\mathtt{e}_1)$
$\mathtt{b}_2 \leftarrow \mathit{not}\,(\mathtt{e}_1), \mathit{not}\,(\mathtt{e}_2), \mathit{not}\,(\mathtt{e}_3)$
$\mathtt{b}_3 \leftarrow \mathtt{e}_1$
$\mathtt{b}_4 \leftarrow \mathtt{e}_1, \mathit{not}\,(\mathtt{e}_5)$
$\mathtt{b}_5 \leftarrow \mathtt{e}_2$
$\mathtt{b}_7 \leftarrow \mathtt{e}_5, \mathit{not}\,(\mathtt{e}_8)$
$\leftarrow \mathtt{b}_1, \mathtt{b}_2$
$\leftarrow \mathtt{b}_2$
$\leftarrow \mathtt{b}_3$
$\leftarrow \mathtt{b}_4$
$\leftarrow \mathtt{b}_5$
$\leftarrow \mathtt{b}_7$

Figure 7.1: Deadlock translation example.

Proof of Theorem 51:
Now we have all the ingredients needed to prove Theorem 51. 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 $\mathit{Mark}(C) = M$, and for every transition t enabled in M there exists a configuration $C \cup \{e\}$ such that $e \notin C$ and $h(e) = t$. Clearly this also holds for all reachable deadlock markings. The finite complete prefix β will thus have a configuration C with no cut-off events, which can not be extended by any event $e \in E$, iff Σ has a reachable deadlock marking. Now Lemma 56 has shown a one-to-one correspondence between deadlock configurations without cut-off events, and stable models of $P_D(\beta)$. Therefore $P_D(\beta)$ will have a stable model iff Σ has a reachable deadlock marking. It also holds by Lemma 56 that for any stable model Δ of $P_D(\beta)$, $C = \mathit{Events}(\Delta)$ is a deadlock configuration of β, such that $\mathit{Mark}(C)$ a reachable deadlock marking of Σ. $\square$

In Fig. 7.1 an example of the deadlock translation is given. The prefix $N2$ is a finite complete prefix of the 1-safe nets system $N1$. The cut-off events of $N2$ are marked with crosses. The translated program $P_D(N2)$ has only one stable model $\Delta = \{\mathtt{be}_1, \mathtt{be}_2, \mathtt{e}_3, \mathtt{be}_5, \mathtt{be}_8, \mathtt{b}_1\}$, and the set $\mathit{Events}(\Delta) = \{e_3\}$ is a deadlock configuration of $N2$.

7.3 REACHABILITY TRANSLATION

Next we will present a way of translating reachability problems. First we need a way of making statements about an individual marking.

Definition 57 *An* assertion *on a marking of a 1-safe net system* Σ *is a tuple* $\langle S^+, S^- \rangle$*, where* $S^+, S^- \subseteq S$*, and* $S^+ \cap S^- = \emptyset$*. The assertion* $\langle S^+, S^- \rangle$ agrees *with a marking* M *of* Σ *iff:*

$$S^+ \subseteq \{s \in S \mid M(s) = 1\} \wedge S^- \subseteq \{s \in S \mid M(s) = 0\}.$$

With assertions we can easily formulate both the reachability and submarking reachability problems. The idea is again to add some integrity rules to the program which remove all stable models of $P_B(\beta)$ which do not agree with the assertion. The basic structure is the same as for deadlocks, only a set of atoms which represent the marking of the original net are added.

Definition 58 *Let* $\beta = \langle\langle B, E, F\rangle, h\rangle$ *be a finite complete prefix of a given 1-safe net system* $\Sigma = \langle S, T, F, M_0\rangle$*, and let* $\phi = \langle S^+, S^- \rangle$ *be an assertion on the places of* Σ*. Let* $P_R(\beta, \phi)$ *be a logic program containing all the rules of the program* $P_B(\beta)$ *of Definition 41, and also the following rules:*

1. *For all* $b_i \in \{b_j \in B \mid h(b_j) \in S^+ \cup S^- \wedge {}^\bullet b_j \in E \setminus \mathit{Cutoffs}(\beta)\}$ *a rule:*
   $\mathtt{b_i} \leftarrow \mathtt{e_l}, \mathit{not}\,(\mathtt{e_{p_1}}), \ldots, \mathit{not}\,(\mathtt{e_{p_n}}),$
   such that $\{e_l\} = {}^\bullet b_i$*, and* $\{e_{p_1}, \ldots, e_{p_n}\} = b_i^\bullet \setminus \mathit{CutOffs}(\beta)$*.*

2. *For all* $b_i \in \{b_j \in B \mid h(b_j) \in S^+ \cup S^- \wedge {}^\bullet b_j \in E \setminus \mathit{Cutoffs}(\beta)\}$ *a rule:*
   $\mathtt{s_i} \leftarrow \mathtt{b_i},$
   such that $s_i = h(b_i)$*.*

3. *For all* $s_i \in S^+$ *a rule:*
   $\leftarrow \mathit{not}\,(\mathtt{s_i}).$

4. *For all* $s_i \in S^-$ *a rule:*
   $\leftarrow \mathtt{s_i}.$

Note that only conditions of the prefix β and places of Σ which can affect the assertion ϕ are translated. Also cut-off postset conditions are not translated, because cut-offs will not be fired.

Theorem 59 *Let* β *be a finite complete prefix of a 1-safe net system* Σ*, and let* $P_R(\beta, \phi)$ *be a logic program translation by Definition 58. The logic program* $P_R(\beta, \phi)$ *has a stable model iff there exists a reachable marking of* Σ *which agrees with* ϕ*. Additionally, for any stable model* Δ *of* $P_R(\beta, \phi)$*, the configuration* $C = \mathit{Events}(\Delta)$ *is a configuration of* β*, such that* $\mathit{Mark}(C)$ *is a reachable marking of* Σ *which agrees with* ϕ*.*

We prove the Theorem 59 by stepwise adding rules to the base program $P_B(\beta)$.

Lemma 60 *Let program $P_P(\beta, \phi)$ be a program made by adding rules of the type 1 of Definition 58 to the program $P_B(\beta)$. The program $P_P(\beta, \phi)$ has the same number of stable models as $P_B(\beta)$, and the stable models agree on the set of atoms $Atoms(P_B(\beta))$.*

Proof:
The proof is immediate by Lemma 52. □

Lemma 61 *For any stable model Δ of $P_P(\beta, \phi)$ and for any condition atom $\mathtt{b_i} \in Atoms(P_P(\beta, \phi))$ it holds that $\mathtt{b_i} \in \Delta$ iff $b_i \in Cut(Events(\Delta))$.*

Proof:
Identical to the proof of Lemma 55 when $P_C(\beta)$ is replaced by $P_P(\beta, \phi)$. □

Lemma 62 *Let program $P_Q(\beta, \phi)$ be a program made by adding rules of the type 2 of Definition 58 to the program $P_P(\beta, \phi)$. The program $P_Q(\beta, \phi)$ has the same number of stable models as $P_P(\beta, \phi)$, and the stable models agree on the set of atoms $Atoms(P_P(\beta, \phi))$.*

Proof:
The proof is immediate by Lemma 52. □

Lemma 63 *For any stable model Δ of $P_Q(\beta, \phi)$ and for any place atom $\mathtt{s_i} \in Atoms(P_Q(\beta, \phi))$ it holds that $\mathtt{s_i} \in \Delta$ iff $s_i \in Mark(Events(\Delta))$.*

Proof:
Fix any stable model Δ of $P_Q(\beta, \phi)$, and any atom $\mathtt{s_i}$ of the program. Now the rules corresponding to this atom are all of the form: $\mathtt{s_i} \leftarrow \mathtt{b_i}$, such that $s_i = h(b_i)$. Now clearly by Lemma 52: $\mathtt{s_i} \in \Delta$ iff $\mathtt{b_i} \in \Delta$ for some condition b_i for which $s_i = h(b_i)$, which combined with Lemma 61 implies the claim. □

Proof of Theorem 59:
We can now prove Theorem 59. 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 $Mark(C) = M$. The stable models of the program $P_R(\beta, \phi)$ are always a subset of the stable models of the program $P_Q(\beta, \phi)$, because only integrity rules of type 3 and 4 of Definition 58 have been added. By Lemma 62 there exists a stable model of $P_Q(\beta, \phi)$ corresponding to each configuration of the prefix β which contains no cut-off events. Also Lemma 63 shows that the atoms $\mathtt{s_i}$ reflect the corresponding marking of Σ. There are now two cases left to prove:

- If Σ has a reachable marking M which agrees with ϕ, then by Lemma 62 there exists a stable model Δ of $P_Q(\beta, \phi)$, such that it holds that $Mark(Events(\Delta)) = M$. Because M agrees with ϕ, it directly holds by Lemma 63 that for all $\{\mathtt{s_i} \mid s_i \in S^+\}$: $\mathtt{s_i} \in \Delta$, and also for all $\{\mathtt{s_j} \mid s_j \in S^-\}$: $\mathtt{s_j} \in \overline{\Delta}$. Therefore there is no integrity rule in $P_R(\beta, \phi)$ which can be used, which implies Δ is also a stable model of $P_R(\beta, \phi)$.

- If Σ has a reachable marking M which does not agree with ϕ, then by Lemma 62 there exists a stable model Δ of $P_Q(\beta, \phi)$, such that $Mark(Events(\Delta)) = M$. Because M does not agree with ϕ, it holds by Lemma 63 that either there exists a place s_i in S^+ such that $\mathtt{s_i} \in \overline{\Delta}$, or there exists a place s_j in S^- such that $\mathtt{s_j} \in \Delta$. In the first case an integrity rule of type 3, and in the second case an integrity rule of type 4 implies that Δ is not a stable model of $P_R(\beta, \phi)$.

This concludes the proof of Theorem 59. □

It is easy to see that the sizes of all the translations presented are linear in the size of the prefix β, i.e. $\mathcal{O}(|B| + |E| + |F|)$. Because the rule-based constraint programming system we use needs linear space in the size of the input program, deadlock and reachability property checking exploiting these translations can be made using linear space in the size of the prefix. The translations are also local, which makes them straightforward to implement using linear time in the size of the prefix.

8 IMPLEMENTATION

We have implemented both the deadlock and reachability checking translations in a tool called `mcsmodels`. For deadlock checking we have made extensive benchmarking against other finite complete prefix based deadlock checking algorithms. For reachability checking we have run a smaller number of experiments to show the feasibility of the approach, but have not conducted benchmarking against other reachability checkers.

The `mcsmodels` reads a file containing the description of a finite complete prefix generated by the ERVunfold algorithm [10], a part of PEP-toolset [15]. It then generates a logic program using the deadlock or reachability translation, which is then through an internal interface given to the `smodels` stable model generator. After the translation has been created, the `smodels` computational engine is used to check whether a stable model of the program exists. If one exists, the program outputs a sequence of transitions which leads to a deadlock or to a witness/counterexample state, using the found stable model.

8.1 DEADLOCK CHECKING

The deadlock checking implementation contains a number of optimizations, which improve the efficiency of the used translation. Let us define *enabledness closure*, which is used in the optimizations. The intuition behind the enabledness closure is that given a set of events as an input, it generates a set of events which can "directly influence" the enabling of some event in the input set.

Definition 64 *Let $\beta = \langle\langle B, E, F\rangle, h\rangle$ be a prefix of the unfolding of the net system Σ, and let $E' \subseteq E$. We define the enabledness closure $EC(E')$ as $EC_{|E|}$ of the following inductive definition:*

$$EC_0 = E'$$

$$EC_{i+1} = EC_i \cup {}^\bullet({}^\bullet EC_i) \cup ({}^\bullet EC_i)^\bullet$$

Actually what we have defined above is that $EC(E')$ is the least fixpoint of a function f from $\mathcal{P}(E)$ to $\mathcal{P}(E)$, which is monotonically non-decreasing, i.e. $E' \subseteq f(E')$ for all $E' \subseteq E$. It is easy to see that $EC(E') \subseteq E$, and that for all $e \in EC(E'), e' \in E : e' \leq_F e \Rightarrow e' \in EC(E')$, i.e. that $EC(E')$ is a causally closed set of events. There are also other events, which are included by the part $({}^\bullet EC_i)^\bullet$ of the definition. We define that two events $e, e' \in E$ are in *direct conflict*, denoted $e \#_d e'$ if ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$, i.e. they share preplaces. Let's denote by $\#_d^*$ the reflexive, transitive closure of this relation. It is now also easy to see that for all $e \in EC(E'), e' \in E : e \#_d^* e' \Rightarrow e' \in EC(E')$, i.e. that the set $EC(E')$ is also closed under the direct conflict relation.

The deadlock translation performs the following optimizations:

1. Not generating the program if the number of cut-off events is zero.

2. Removal of blocking of "stubborn" transitions: If we find an event e_i such that $({}^\bullet e_i)^\bullet \setminus \mathit{Cutoffs}(\beta) = \{e_i\}$, the corresponding rule of type 1 of the program $P_B(\beta)$ is replaced by a rule of the form: $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, and the rule 2 of the form: $\mathtt{be_i} \leftarrow \mathit{not}\,(\mathtt{e_i})$ is not created. Also the corresponding liveness rule of type 2 of the program $P_D(\beta)$ of the form: $\leftarrow \mathtt{b_{p_1}}, \ldots, \mathtt{b_{p_n}}$ does not need to be created as far as the event e_i is concerned.

3. Removal of redundant condition rules: The rule of type 1 of the program $P_D(\beta)$ corresponding to condition b_i is removed if the atom $\mathtt{b_i}$ is does not appear elsewhere in the program.

4. Removal of redundant atoms: If a rule of the form: $\mathtt{a_1} \leftarrow \mathtt{a_2}$ would be generated, and this is the only rule in which $\mathtt{a_1}$ appears as a head, then all instances of $\mathtt{a_1}$ are replaced by $\mathtt{a_2}$, and the rule is discarded.

5. Duplicate rule removal: Only one copy of each rule is generated.

6. Enabledness closure: All events not in $EC(\mathit{CutOffs}(\beta))$ are removed from the prefix β together with their postset conditions.

For the optimization 1 it is easy to see that the net system Σ will deadlock, because the finite complete prefix is finite and does not contain any cut-offs. Thus the net system Σ can fire only a finite number of transitions. It also is straightforward to prove that the optimizations 3-5 do not alter the number of stable models the program has. The optimization 2 is motivated by stubborn sets [39]. The intuition is that whenever e_i is enabled, it must be disabled in order to reach a deadlock. However, the only way of disabling e_i is to fire it. Therefore we can discard all configurations in which e_i is enabled as not being deadlock configurations.

We argue that optimization 2 is correct, i.e. the stable models of the program $P_D(\beta)$ are not affected by it (modulo the possible removal of the atom $\mathtt{be_i}$ from the set of atoms of the optimized program). Consider the original program, and an optimized one in which an event e_i has been optimized using optimization 2. If we look only at the two programs without the deadlock detection parts added by Definition 50, their only difference is that in the original program it is possible to leave the event e_i enabled but not fired, while this is not possible in the optimized program. Thus clearly the set of stable models of the optimized program is a subset of the stable models of the original one. If we have any configuration in which the event e_i is enabled but is not fired, then the set of atoms corresponding to this configuration is not a stable model of the original program. This is the case because the integrity rule of type 2 of Definition 50 corresponding to the event e_i eliminates such a potential stable model. Therefore the optimized program will have the same number of stable models as the original one.

The optimization 6 is inspired by the McMillan's deadlock checking algorithm [28, 30]. In his algorithm some events of the prefix are calculated to be so called *spoiler events*. McMillan's algorithm then tries to construct a configuration containing only spoiler events, which is in conflict with all the cut-off events. If such a configuration exists, then the net has a deadlock, otherwise the net is deadlock free. To explain why this is the case, consider a configuration C containing only spoiler events, which is in conflict with all cut-off events. No execution starting from configuration C can be infinite, because to create an infinite execution one needs to fire a cut-off event. Therefore all execution starting from C will deadlock. Also if we have any deadlock configuration C', then clearly C' is in conflict with all the cut-off events. The spoilers are calculated in McMillan's approach in a way which guarantees that for any such C' there is some configuration C such that C contains only spoilers, and that $C \subseteq C'$.

The spoilers of a cut-off event e are $Spoilers(e) = \{e' \in E$: there exists $e'' \in [e]$, such that: $e' \#_d e'' \wedge e' \notin [e]\}$, i.e. events which are in direct conflict with the local configuration of a cut-off event, but are not in the local configuration. The set of spoilers of a net is then the union of all the sets of cut-off spoiler events. Firing any spoiler event guarantees that the corresponding cut-off event can not become enabled. It is easy to see that no spoiler event of any cut-off event is removed by the optimization 6, because the set $EC(\mathit{CutOffs}(\beta))$ contains all events which are in direct conflict with the local configuration of any cut-off event.

Now we proceed to prove that optimization 6 is correct. Consider first the (maximal) unfolding of the net system. Clearly the unfolding has a deadlock iff the net system has a deadlock. When we look an events e removed by the optimization 6, it is not in a local configuration of any cut-off. Therefore the event e appears exactly once in the unfolding, and the number of events in the unfolding after e, i.e. the set $\{e' : e \leq_F e'\}$ is finite. Therefore if we remove e together with all the events after it from the unfolding, we will still have an infinite run if the original unfolding had an infinite run. In other words the optimization 6 can not transform an infinite execution into a deadlocking execution of the unfolding. This also holds for the prefix because there is bijective mapping between e and its future between the prefix and the unfolding.

To show a deadlock is preserved if some deadlock exists, consider again the original prefix and deadlock configuration C'. Consider an event e which is a maximal event of the events to be removed by optimization 6, and remove only this maximal event from the prefix, thus creating a partially optimized prefix. Now if $C' \cap \{e\} = \emptyset$, then C' is also a deadlock of the partially optimized prefix. Otherwise $e \in C'$, and we claim that $C = C' \setminus \{e\}$ is a configuration of the partially optimized prefix which is in conflict with all the cut-off events. This is the case because e is not in the transitive direct conflict relation $\#_d^*$ of the local configurations of cut-off events (and thus it's not a spoiler for any cut-off event). Therefore C is also in conflict with all the cut-off events because C' is in conflict with all cut-off events (because it's a deadlock), and only a non-spoiler event e was removed from C'. Because C is in conflict with all the cut-off events, all executions starting from C in the partially optimized prefix will eventually deadlock (i.e. either C is a deadlock or all its maximal extensions are deadlocks). We can now prove that the optimization 6 does not loose the deadlock property by induction on the events removed by optimization 6, proceeding a maximal event of the remaining optimized events at a time.

We have thus proved that optimization 6 on its own is correct. We now argue that it is also compatible with the other optimizations. The only optimization that could interfere with this optimization is optimization 2. The optimization 2 is enabled only when an event e_i is not in conflict with any non-cut-off event. However, because for any event $e_i \in EC(\mathit{CutOffs}(\beta))$ it holds that $({}^\bullet e_i)^\bullet \subseteq EC(\mathit{CutOffs}(\beta))$, the optimization 2 is enabled for an event e_i in the prefix optimized by optimization 6 iff it would be enabled for e_i in the original prefix. Therefore we can first do the optimization 6 without affecting the events optimization 2 is applied to in the optimized prefix, and the optimization 6 is therefore correct.

Notice that the optimization 6 can result in a prefix, whose deadlock configuration is not a deadlock of the original net. However, any execution starting from this configuration will lead to a deadlock. The algorithm will generate a sequence leading to a deadlock using this configuration as a starting point, and then firing enabled events until a deadlock of the original prefix is reached. An interesting point is that the optimization 1 can be seen as a special case of optimization 6.

We do quite an extensive set of optimizations. The optimizations 1, 2, and 6 are deadlock detection specific. The optimizations 3-5 can be seen as general logic program optimizations based on static analysis, and could in principle be done in the stable model generator after the translation. The optimizations 1-4, and 6 are implemented using linear time and space in the size of the prefix. The duplicate rule removal is implemented with hashing.

We use succinct rule encodings with extended rules when possible. The two rules $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}, \mathit{not}\,(\mathtt{be_i})$, and $\mathtt{be_i} \leftarrow \mathit{not}\,(\mathtt{e_i})$ can be more succinctly encoded by an extended rule of the form: $\{\mathtt{e_i}\} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$. Also $\leftarrow 2\{\mathtt{a_1}, \mathtt{a_2}\}$ is replaced by: $\leftarrow \mathtt{a_1}, \mathtt{a_2}$. We also sort the rules after the translation. In our experiments the sorting seems to have only a minimal effect on the total running time, but produces nicer looking logic program (debugging) output.

8.2 DEADLOCK CHECKING EXPERIMENTAL RESULTS

We have made experiments with our approach using examples by Corbett [3], McMillan [28, 30], and Melzer and Römer [32]. They were previously used by Melzer and Römer in [32] and by Best and Römer in [1], where additional information about them can be found. We compare our approach with two other finite complete prefix based deadlock checking methods. The first method is the branch-and-bound deadlock detection algorithm by McMillan [28, 30, 32], and the other is the mixed integer programming approach by Melzer and Römer [32].

The Figures 8.1, 8.3, and 8.5 present the example nets used. The Figures 8.2, 8.4, and 8.6 present the running times in seconds for the algorithms used in this work, and for the mixed integer programming approach those presented in [32]. The times have been measured using a Pentium II 267MHz, 512MB RAM, 128MB swap, Linux 2.2.3, g++ (egcs-2.91.60), `mcsmodels` 1.4 (containing `smodels` 2.23), McMillan's algorithm version 2.1.0 by Stefan Römer, and ERVunfold version 4.5.1. The mixed integer programming approach experiments by Melzer and Römer used a commercial MIP-solver CPLEX, and were conducted on a Sparcstation 20/712, 96MB RAM.

Problem(size)	$\|S\|$	$\|T\|$	$\|B\|$	$\|E\|$	#c	States
DPD(5)	45	45	1582	790	211	3488
DPD(6)	54	54	3786	1892	499	19860
DPD(7)	63	63	8630	4314	1129	109964
DPH(5)	48	67	2712	1351	547	3112
DPH(6)	57	92	14590	7289	3407	16896
DPH(7)	66	121	74558	37272	19207	79926
ELEVATOR(2)	146	299	1562	827	331	1061
ELEVATOR(3)	327	783	7398	3895	1629	7120
ELEVATOR(4)	736	1939	32354	16935	7337	43439
FURNACE(1)	27	37	535	326	189	343
FURNACE(2)	40	65	4573	2767	1750	3777
FURNACE(3)	53	99	30820	18563	12207	30860
RING(7)	91	77	813	403	79	16999
RING(9)	117	99	1599	795	137	211527
RW(9)	48	181	9272	4627	4106	522
RW(12)	63	313	98378	49177	45069	4109

Figure 8.1: Example net prefixes.

Problem(size)	DL	Unf[1]	DC^2_{MIP}	DC^1_{McM}	DC^1_{smo}	DC^1_{smo} -n	CP	CP -n
DPD(5)	N	0.1	17.3	0.9	0.3	0.1	0	201
DPD(6)	N	0.5	82.8	8.2	2.0	0.3	0	546
DPD(7)	N	2.2	652.6	91.5	11.4	0.8	0	1159
DPH(5)	N	0.2	42.9	3.0	0.6	0.1	0	156
DPH(6)	N	4.1	1472.8	542.8	13.8	1.3	0	1561
DPH(7)	N	101.7	-	$vm(2502.9)$	324.9	11.3	0	7839
ELEVATOR(2)	Y	0.1	2.3	0.3	0.2	0.0	2	4
ELEVATOR(3)	Y	1.3	14.5	4.8	4.3	0.2	3	5
ELEVATOR(4)	Y	27.4	387.8	177.0	85.1	1.0	4	6
FURNACE(1)	N	0.0	0.3	0.1	0.0	0.0	0	27
FURNACE(2)	N	0.4	18.1	4.4	0.2	0.1	0	192
FURNACE(3)	N	14.3	1112.5	306.7	2.3	1.1	0	1171
RING(7)	N	0.1	17.1	0.1	0.2	0.0	0	104
RING(9)	N	0.2	71.2	0.3	0.8	0.1	0	244
RW(9)	N	0.5	58.5	30.7	0.2	0.2	0	0
RW(12)	N	25.3	24599.9	$vm(4151.2)$	2.3	2.2	0	0

Figure 8.2: Deadlock checking running times in seconds:
[1] = Pentium II 267MHz, 512MB RAM.
[2] = Sparcstation 20/712, 96MB RAM [32].

The rows of the tables correspond to different problems. The columns represent: sum of user and system times measured by `/usr/bin/time` command, or times reported in [32], depending on the column:

- Unf = time for unfolding (creating a finite complete prefix) (ERVunfold), average of 5 runs, using the option -@@ which enables the new possible extensions algorithm, see [10].
- DC_{MIP} = time for Mixed integer programming approach in [32].
- DC_{McM} = time for McMillan's algorithm, average of 5 runs.
- DC_{smo} = time for `mcsmodels`, average of 5 runs.
- DC_{smo}-n = time for `mcsmodels`, average of 5 runs, using the -n option which disables the lookahead of `smodels`.

The marking $vm(n)$ notes that the program ran out of virtual memory after n seconds. The other fields of the figures are as follows: $|S|$: number of places,

Problem(size)	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert B\rvert$	$\lvert E\rvert$	#c	States
DME(4)	269	196	2381	652	16	$> 10^4$
DME(5)	336	245	4096	1145	25	$> 10^5$
DME(6)	403	294	6451	1830	36	$> 10^6$
DME(7)	470	343	9542	2737	49	$> 10^7$
DME(8)	537	392	13465	3896	64	$> 10^8$
DME(9)	604	441	18316	5337	81	$> 10^9$
DME(10)	671	490	24191	7090	100	$> 10^{10}$
DME(11)	738	539	31186	9185	121	$> 10^{11}$
SYNC(2)	72	88	3884	2091	474	17874
SYNC(3)	106	270	28138	15401	5210	116446

Figure 8.3: Example net prefixes.

Problem(size)	DL	Unf[1]	DC^2_{MIP}	DC^1_{McM}	DC^1_{smo}	DC^1_{smo} -n	CP	CP -n
DME(4)	N	0.5	216.1	0.6	1.4	0.1	0	567
DME(5)	N	1.5	1968.3	2.3	4.8	0.4	0	1392
DME(6)	N	4.2	13678.3	8.4	13.6	1.0	0	3610
DME(7)	N	10.5	-	29.1	32.8	2.4	0	7954
DME(8)	N	23.1	-	84.0	75.0	5.2	0	13939
DME(9)	N	44.8	-	220.2	150.8	13.6	0	27871
DME(10)	N	82.5	-	523.4	282.3	36.3	0	68845
DME(11)	N	142.1	-	1140.6	482.0	100.7	0	154033
SYNC(2)	N	0.6	171.6	11.7	0.5	0.1	0	407
SYNC(3)	N	23.0	11985.0	6164.8	23.2	2.2	0	2747

Figure 8.4: Deadlock checking running times in seconds:
[1] = Pentium II 267MHz, 512MB RAM.
[2] = Sparcstation 20/712, 96MB RAM [32].

$|T|$: number of transitions, $|B|$: number of conditions, $|E|$: number of events, #c: number of cut-off events, States: number of reachable markings, DL: Y - the net system has a deadlock, CP: *choice points* i.e. the size of the search tree of `smodels` during the run, CP -n: number of choice points, when using the nolookahead option. The DC_{smo} columns also includes the logic program translation time, which was always under 10 seconds for the examples.

The logic programming approach using the `smodels` system was able to produce an answer for all the examples presented here, while the McMillan's algorithm implementation ran out of virtual memory on some of the larger examples. Our approach was sometimes much faster, see e.g. FURNACE(3), RW(12), SYNC(3), BDS(1), GASQ(4), and Q(1). The McMillan's algorithm was faster than `mcsmodels` (without -n) on the following problem classes: RING, HART, SENT and SPD. These problems are quite easy for both methods, running times for the first three were a few seconds, and for the fourth still under 6 seconds. On the DME and KEY examples our approach is scaling better as the problem sizes increase. McMillan's algorithm is most competitive when the number of cut-off events is relatively small.

The option -n has a dramatic effect on reducing the running times of the examples. The option disables the lookahead heuristic of `smodels`, which is designed to minimize the number of choice points, see the difference between columns CP and CP -n. However, the number of choice points for these examples is quite low, and therefore for this set of examples the benefit obtained from this heuristic does not cover the computational effort needed to obtain it. On the largest DME examples the -n version seems to be scaling worse, however it it still almost five times faster than the version doing

Problem(size)	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert B\rvert$	$\lvert E\rvert$	#c	States
BDS(1)	53	59	12310	6330	3701	36096
FTP(1)	176	529	178085	89046	35197	104910
GASN(3)	143	223	2409	1205	401	1769
GASN(4)	258	465	15928	7965	2876	14846
GASN(5)	428	841	100527	50265	18751	115183
GASQ(3)	284	475	2593	1297	490	1704
GASQ(4)	1428	2705	19864	9933	4060	15430
OVER(4)	71	74	1536	783	237	4174
OVER(5)	90	95	7266	3697	1232	33459
HART(50)	252	152	354	202	1	102
HART(75)	377	227	529	302	1	152
HART(100)	502	302	704	402	1	202
KEY(2)	94	92	1310	653	199	536
KEY(3)	129	133	13941	6968	2911	4923
KEY(4)	164	174	135914	67954	32049	44819
MMGT(3)	122	172	11575	5841	2529	7702
MMGT(4)	158	232	92940	46902	20957	66308
Q(1)	163	194	16123	8417	1188	123596
SENT(75)	54	105	533	266	40	331
SENT(100)	329	130	608	291	40	381
SPD(1)	33	39	4929	2882	1219	8689

Figure 8.5: Example net prefixes.

lookahead in DME(11).

We do not have access to the MIP-solver used in [32], our experiments with the McMillan's algorithm implementation seem to indicate that the computer we made our experiments on is approximately five times faster than theirs. This makes it difficult to comment on the absolute running times between different machines. However, our approach is scaling better on most examples, see e.g. RW, DME, and SYNC examples.

An observation that should be made is that the number of choice points for `smodels` (without -n) in these examples is very low, with a maximum of 9 choice points in the example KEY(4). This means that on this example set the search space pruning techniques were very effective in minimizing the number of nondeterministic choices that were needed to solve the examples.

8.3 REACHABILITY CHECKING

We perform similar optimizations for reachability checking as we did for deadlock checking. However, these optimizations need to be modified a little bit to work for reachability checking.

In the reachability translation we call an event *visible* if its pre- or postsets contains some condition which is mapped to some place in the reachability assertion. All other events are called *invisible*. More formally:

Definition 65 *Let $\phi = \langle S^+, S^- \rangle$ be an assertion on the places of 1-safe net system Σ, and let $\beta = \langle \langle B, E, F \rangle, h \rangle$ be a prefix of the unfolding of Σ. The set of* visible events *$Visible(\beta, \phi)$ is defined as:*

$$\{e \in E \mid h({}^\bullet e \cup e^\bullet) \cap (S^+ \cup S^-) \neq \emptyset\}$$

The reachability translation performs the following optimizations:

Problem(size)	DL	Unf[1]	$\mathrm{DC}^1_{\mathrm{McM}}$	$\mathrm{DC}^1_{\mathrm{smo}}$	$\mathrm{DC}^1_{\mathrm{smo}}$ -n	CP	CP -n
BDS(1)	N	2.4	94.7	1.4	0.4	0	73
FTP(1)	N	950.9	$vm(7032.6)$	702.9	9.4	0	590
GASN(3)	N	0.2	6.0	0.8	0.1	0	528
GASN(4)	N	8.2	1217.0	41.9	2.3	0	4817
GASN(5)	N	401.3	$vm(4697.5)$	1678.3	62.6	0	37241
GASQ(3)	N	0.3	4.8	0.8	0.1	0	566
GASQ(4)	N	14.1	1794.7	51.5	2.8	0	4511
OVER(4)	N	0.1	0.5	0.1	0.0	0	45
OVER(5)	N	1.8	22.7	0.3	0.3	0	113
HART(50)	Y	0.1	0.0	0.1	0.0	5	1
HART(75)	Y	0.1	0.0	0.1	0.0	6	1
HART(100)	Y	0.2	0.0	0.2	0.0	6	1
KEY(2)	Y	0.1	0.2	0.2	0.0	4	107
KEY(3)	Y	4.8	22.0	17.8	0.5	6	312
KEY(4)	Y	398.3	$vm(5369.6)$	1287.2	5.7	9	2134
MMGT(3)	Y	3.3	335.3	6.1	0.6	0	1490
MMGT(4)	Y	308.5	$vm(4998.8)$	523.4	16.5	0	13179
Q(1)	Y	14.8	37.1	7.7	0.5	6	9
SENT(75)	Y	0.1	0.0	0.0	0.0	4	30
SENT(100)	Y	0.1	0.0	0.0	0.0	4	30
SPD(1)	Y	0.7	3.4	5.1	0.1	9	60

Figure 8.6: Deadlock checking running times in seconds:
[1] = Pentium II 267MHz, 512MB RAM.

1. Removal of blocking of invisible "stubborn" transitions: If we find an event e_i such that $({}^\bullet e_i)^\bullet \setminus \mathit{Cutoffs}(\beta) = \{e_i\}$ and $e_i \notin \mathit{Visible}(\beta, \phi)$, the corresponding rule of type 1 of the program $P_B(\beta)$ is replaced by a rule of the form: $\mathtt{e_i} \leftarrow \mathtt{e_{p_1}}, \ldots, \mathtt{e_{p_n}}$, and the rule 2 of the form: $\mathtt{be_i} \leftarrow$ *not* $(\mathtt{e_i})$ is not created.

2. Removal of redundant condition rules: The rule of type 1 of the program $P_R(\beta, \phi)$ corresponding to condition b_i is removed if the atom $\mathtt{b_i}$ is does not appear elsewhere in the program.

3. Removal of redundant atoms: If a rule of the form: $\mathtt{a_1} \leftarrow \mathtt{a_2}$ would be generated, and this is the only rule in which $\mathtt{a_1}$ appears as a head, then all instances of $\mathtt{a_1}$ are replaced by $\mathtt{a_2}$, and the rule is discarded.

4. Duplicate rule removal: Only one copy of each rule is generated.

5. Enabledness closure: All events not in $EC(\mathit{Visible}(\beta, \phi))$ are removed from the prefix β together with their postset conditions.

The optimizations 2-4 are not reachability checking specific, and are implemented in the same way as for deadlock checking. The optimization 1 is now limited to only invisible events instead of being applied to any event. The correctness argument is similar as for the deadlock detection case. After the optimization of an event e_i the optimized base program $P_B(\beta)$ has potentially less stable models, namely those stable models have been removed in which e_i is enabled but is not fired. We now only need to consider only those stable models of the non-optimized program in which e_i is enabled but is not fired. Let Δ now be any such stable model of the non-optimized $P_B(\beta)$ which is also a stable model of the non-optimized reachability program $P_R(\beta, \phi)$, and let $C = \mathit{Events}(\Delta)$ be the corresponding configuration. Now because Δ is a stable model, then also $\Delta' = \mathit{Model}(C \cup \{e_i\})$ is a stable model of the optimized program because e_i is enabled and invisible and thus doesn't affect any of the places mentioned in the assertion ϕ. The proof in the case when Δ is not a stable model of the non-optimized $P_R(\beta, \phi)$ is done in the same way.

For the optimization 5 we need to do a more complicated proof.

Lemma 66 *Let β' be the prefix of β in which all of the events not in the set $EC(\mathit{Visible}(\beta, \phi))$ have been removed together with their postset events. The prefix β' has a configuration C', such that $\mathit{Mark}(C')$ agrees with ϕ iff the prefix β has a configuration C, such that $\mathit{Mark}(C)$ agrees with ϕ.*

Proof:
Because $EC(\mathit{Visible}(\beta, \phi))$ is a causally closed set of events, it is easy to see that β' really is a prefix of β. Let's now start inspecting the configurations.

One direction is trivial: Let C' be any configuration of β', then $C = C'$ is also a configuration of β and we are done.

In the other direction: Let C be a configuration of β. We claim that $C' = C \cap EC(\mathit{Visible}(\beta, \phi))$ is a configuration of β', such that for all places mentioned in the assertion ϕ the markings have the same number of tokens. It is straightforward to prove by induction that the intersection of a configuration and a causally closed set of events is also a configuration, as it is a union of local configurations of events which are in pairwise *co* relation. Therefore, because C is a configuration, and $EC(\mathit{Visible}(\beta, \phi))$ is a causally closed set of events, $C' = C \cap EC(\mathit{Visible}(\beta, \phi))$ is also a configuration. Because $EC(\mathit{Visible}(\beta, \phi))$ contains all visible events, no event in $C \setminus C'$ is visible, and thus for all places mentioned in the assertion ϕ the markings $\mathit{Mark}(C')$ and $\mathit{Mark}(C)$ have the same number of tokens, and therefore $\mathit{Mark}(C')$ agrees with ϕ iff $\mathit{Mark}(C)$ agrees with ϕ. $\square$

We have thus proved that optimization 5 on its own is correct. We now argue that it is also compatible with the other optimizations. The only optimization that could interfere with this optimization is optimization 1. The optimization 1 is enabled only when an event e_i is not in conflict with any non-cut-off event. However, because for any event $e_i \in EC(\mathit{Visible}(\beta, \phi))$ it holds that $({}^\bullet e_i)^\bullet \subseteq EC(\mathit{Visible}(\beta, \phi))$, the optimization 1 is enabled for an event e_i in the prefix optimized by optimization 5 iff it would be enabled for e_i in the original prefix. Therefore we can first do the optimization 5 without affecting the events optimization 1 is applied to in the optimized prefix, and the optimization 5 is therefore correct.

For reachability checking we have expanded the set of properties beyond simple reachability assertions. We allow checking for assertions, which are arbitrary Boolean expressions using the places of the original net as variables. If the Boolean expression evaluates to true in a marking, the marking satisfies the assertion specified by the Boolean expression. It is easy to see that checking for such assertions is still in NP, and we have implemented this feature by a simple (linear size in the length of the expression) logic program which evaluates the truth value of the expression when the places have been given a valuation, and removes all stable models in which the expression evaluates to false.

8.4 REACHABILITY CHECKING EXPERIMENTAL RESULTS

For reachability checking we have not benchmarked our approach against other finite complete prefix based approaches. We used the same Pentium II 267 MHz and software configuration to run the experiments as with the deadlock checking examples. The results are of similar nature, in most cases disabling the lookahead option of `smodels` improves performance.

The columns in Figures 8.7-8.9 are the same as for deadlock checking examples, the only new column is Reach: Y - the checked marking is reachable. Most of the checked reachability properties were generated by choosing a random marking from the interleaved state space, and then creating an assertion which requires that all places marked in that marking should contain a token. Therefore the checked properties are submarking reachability properties, and any marking which has tokens on (at least) all the places mentioned in the assertion agrees with the assertion. The markings were created by Claus Schröter.

Problem(size)	Reach	DC_{smo}	DC_{smo} -n	CP	CP -n
DPD(5)	Y	0.3	0.0	0	32
DPD(6)	Y	1.7	0.1	0	50
DPD(7)	Y	0.3	0.3	1	33
DPH(5)	Y	0.6	0.1	0	64
DPH(6)	Y	22.2	0.5	0	74
DPH(7)	Y	826.5	8.3	1	2807
ELEVATOR(2)	Y	0.1	0.0	0	95
ELEVATOR(3)	Y	1.8	0.2	0	313
ELEVATOR(4)	Y	37.4	1.5	0	968
FURNACE(1)	Y	0.0	0.0	1	70
FURNACE(2)	Y	0.9	0.1	1	125
FURNACE(3)	Y	54.6	1.7	1	1148
RING(7)	Y	0.1	0.0	1	31
RING(9)	Y	0.0	0.0	0	14
RW(9)	Y	0.1	0.1	0	2
RW(12)	Y	2.0	2.0	0	3
DME(4)	Y	0.1	0.1	0	37
DME(5)	Y	0.2	0.2	0	55
DME(6)	Y	0.4	0.4	0	100
DME(7)	Y	0.6	0.6	0	116
SYNC(2)	Y	0.9	0.1	1	185
SYNC(3)	Y	5.1	0.7	1	85
OVER(4)	Y	0.2	0.0	2	48
OVER(5)	Y	3.9	0.2	3	56
KEY(2)	Y	0.1	0.0	0	106
KEY(3)	Y	14.5	1.2	0	848

Figure 8.7: Reachability checking running times in seconds.

The buffer examples in Figures 8.8 and 8.9 are a very abstract model of a distributed buffer. This example is currently the only example we have in which the lookahead heuristics of `smodels` become effective on some reachability checking problems. In this benchmark the checked markings are hand picked using the knowledge of invariants the net has. Deadlock checking for these buffer examples is very fast with or without lookahead, and that's why they weren't included in the last section.

When comparing the reachability checking running times against the deadlock checking running times, sometimes the deadlock checking takes more time, and sometimes reachability checking is more time consuming. There seems to be some correlation though, in many cases nets which re-

Problem(size)	$\|S\|$	$\|T\|$	$\|B\|$	$\|E\|$	#c	States
BUF(10)	20	11	111	56	1	2^{10}
BUF(20)	40	21	421	211	1	2^{20}
BUF(30)	60	31	931	466	1	2^{30}
BUF(40)	80	41	1641	821	1	2^{40}
BUF(50)	100	51	2551	1276	1	2^{50}
BUF(60)	120	61	3661	1831	1	2^{60}
BUF(70)	140	71	4971	2486	1	2^{70}
BUF(80)	160	81	6481	3241	1	2^{80}
BUF(90)	180	91	8191	4096	1	2^{90}
BUF(100)	200	101	10101	5051	1	2^{100}

Figure 8.8: Buffer example prefixes.

Problem(size)	Reach	Unf	DC_{smo}	DC_{smo} -n	CP	CP -n
BUF(10)	Y	0.0	0.0	0.0	0	54
BUF(20)	Y	0.0	0.1	0.1	0	306
BUF(30)	Y	0.2	0.5	0.3	0	913
BUF(40)	Y	0.6	1.7	0.8	0	2294
BUF(50)	Y	1.5	4.5	2.2	0	6568
BUF(60)	Y	3.5	10.4	5.0	0	15196
BUF(70)	Y	7.3	21.0	12.3	0	86107
BUF(80)	Y	14.1	38.4	25.1	0	230334
BUF(90)	Y	24.7	65.1	65.1	0	1066300
BUF(100)	Y	40.4	104.1	224.6	0	5289913
BUF(10)	N	0.0	0.0	0.0	0	40
BUF(20)	N	0.0	0.1	0.1	0	180
BUF(30)	N	0.2	0.3	0.2	0	412
BUF(40)	N	0.6	0.8	0.8	0	748
BUF(50)	N	1.5	2.1	2.0	0	1184
BUF(60)	N	3.5	4.9	4.4	0	1715
BUF(70)	N	7.3	9.6	9.1	0	2356
BUF(80)	N	14.1	18.1	16.6	0	3087
BUF(90)	N	24.7	30.5	29.0	0	3928
BUF(100)	N	40.4	49.1	45.0	0	4867

Figure 8.9: Buffer example running times in seconds.

quire a lot of time for reachability checking require also a lot of time for deadlock checking and vice versa. However, the buffer example seems to be an exception to this rule.

For reachability checking there are a lot more variants in the set of reachability properties one could check. A more complete set of standard benchmarks for reachability checking would be quite useful.

The deadlock checking example nets, the reachability checking nets and formulas, as well as the C++ source code for `mcsmodels` are available on the Internet at the following addresses:

```
http://www.tcs.hut.fi/~kepa/experiments/tacas99/
http://www.tcs.hut.fi/~kepa/experiments/reach99/
http://www.tcs.hut.fi/~kepa/tools/mcsmodels/
```

9 CONCLUSIONS

Using finite complete prefixes of net unfoldings as a way of alleviating the state space explosion problem in deadlock and reachability checking was quite successful. Our approach was able to deadlock check examples which were not possible by the other benchmarked approaches. The research on using finite complete prefixes in verification has still a lot of open problems, both of theoretical and practical nature. In this work we have tried to give answers to some of these open problems.

The main contributions of this work are: (i) The possible extensions calculation subroutine of the prefix generation algorithm is proved NP-complete. (ii) Model checking a fixed size CTL formula with finite complete prefixes is proved PSPACE-complete. (iii) The translations of the problems of deadlock and reachability checking using finite complete prefixes into the problem of finding a stable model of a logic program are devised. (iv) The implementation of the above mentioned translations in the `mcsmodels` tool is presented, with experimental results supporting the feasibility of the approach.

The reachability and deadlock checker `mcsmodels` reads a file containing the description of a finite complete prefix generated by the ERVunfold algorithm [10]. It then generates a logic program using the deadlock or reachability translation, which is then given to the `smodels` stable model generator. The `smodels` computational engine is used to check whether a stable model of the program exists. If one exists, the program outputs a sequence of transitions which leads to a deadlock or to a witness/counterexample state, using the found stable model. This division of concerns between the translator and the NP-solver made it easier for us to make the `mcsmodels` tool. Also the effort put into the `smodels` tool was vital to reach the level of performance we got. Our experiments show that the approach seems to be quite robust and competitive on the examples available to us. However, a larger set of benchmarks is quite badly needed for the deadlock checking and especially the reachability checking problem to better understand the performance of different algorithms.

There are still some open question related to deadlock and reachability checking using finite complete prefixes. It seems possible to extend the translations to allow for a larger class of acyclic Petri nets to be translated, while still keeping the problem NP-complete. McMillan's algorithm can be seen to be more goal directed algorithm than our approach, and an alternative translation using the basic ideas of McMillan's algorithm could be created. The `smodels` system is quite a general purpose constraint propagation based search engine. Creating specialized algorithms for the two problems at hand could further improve the competitiveness of our approach.

There are several interesting topics for future research. We believe that our NP-completeness complexity result for the possible extensions calculation would suggest that using NP-solvers for possible extension generation is a possibility. The viability of such an approach needs to be investigated.

The creation of a minimal finite complete prefixes is an open question, and of theoretical interest. Our work on prefix minimization [19] is a starting point in that direction. The current definitions of net unfolding and finite complete prefix generation are very algorithmic. An interesting research topic is whether the step semantics of Petri nets could be used as a way of redefining the results of these algorithms in a more abstract and (hopefully) more intuitive way.

Our PSPACE-completeness complexity result of model checking suggests that using the current semantic definition of a finite complete prefix will result in high model checking complexity in the size of the prefix. The finite complete prefixes are the first symbolic representation we know of where there is a difference between the computational complexity of reachability (NP-complete) and repeated reachability (PSPACE-complete) problems. One further research topic is the relation between our complexity result and the work of Graves [16].

We see the possibility of using an alternative semantic definition of a finite complete prefix, which depends both on the net system and the property to be checked. The model checking algorithm for such a "product prefix" would then be potentially simpler and computationally less demanding in the size of the prefix. Seen from another perspective, such an alternative prefix definition will for some net systems and checked properties create larger prefixes than the current one. At least for the safety model checking case such an alternative prefix definition is straightforward to create, the implementation is left for future work. The LTL model checking case seems to be more challenging, and pursuing it is currently work in progress. The drawback with the product prefix approach is that the prefix definition and correctness proofs of the approach need to be redone for each specification formalism used.

The main open problem concerning finite complete prefixes is the creation of efficient model checking algorithms for temporal logics such as LTL and CTL. There are currently many different possible ways of solving these problems using prefixes, each with their own positive and negative sides. Both theoretical and practical work is needed in this area to compare these different approaches, and to implement the ones which help us in fighting the state explosion problem.

Bibliography

[1] E. Best. Partial order verification with PEP. In G. Holzmann, D. Peled, and V. Pratt, editors, *Proceedings of POMIV'96, Workshop on Partial Order Methods in Verification*. American Mathematical Society, 1996.

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

[3] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. Technical report, Department of Information and Computer Science, University of Hawaii at Manoa, 1995.

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

[5] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[6] J. Engelfriet. Branching processes of Petri nets. In *Acta Informatica 28*, pages 575–591, 1991.

[7] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994.

[8] 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.

[9] J. Esparza and M. Nielsen. Decidability issues for Petri Nets - a survey. *Journal of Information Processing and Cybernetics 30(3)*, pages 143–160, 1994.

[10] 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. Invited paper, LNCS 1664.

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

[12] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. Unpublished manuscript, submitted to Formal Methods in System Design, 1998.

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

[14] 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, Seattle, USA, 1988. The MIT Press.

[15] B. Grahlmann. The PEP Tool. In *Proceeding of 9th International Conference on Computer Aided Verification (CAV'97)*, pages 440–443. Springer-Verlag, 1997. LNCS 1254.

[16] 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.

[17] S. Haar. Branching processes of general S/T-systems. In *Proceedings of the Workshop Concurrency, Specification & Programming 1998*, Informatik-Bericht Nr. 110, pages 88–97. Humboldt-University, Berlin, 1998.

[18] 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, 1998.

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

[20] 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.

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

[22] N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977.

[23] M. Kishinewsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practise of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[24] A. Kondratyev, M. Kishinevsky, A Taubin, J. Cortadella, and L. Lavagno. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems, and Computers*, 8(1):67–118, 1998.

[25] 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. Spriger-Verlag, 1999. LNCS 1663.

[26] H. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1982.

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

[28] K. L. McMillan. Using unfoldings to avoid the state space explosion problem in the verification of asynchronous circuits. In *Proceeding of 4th Workshop on Computer Aided Verification (CAV'92)*, pages 164–174, 1992. LNCS 663.

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

[30] K. L. McMillan. A technique of a state space search based on unfolding. *Formal Methods is System Design*, 6(1):45–65, 1995.

[31] K. L. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. In *Proceeding of 7th International Conference on Computer Aided Verification (CAV'95)*, pages 180–195. Springer-Verlag, 1995. LNCS 939.

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

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

[34] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 1999. Accepted for publication.

[35] 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, Dagstuhl, Germany, 1997. Springer-Verlag.

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

[37] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report A47, Helsinki University of Technology, Espoo, Finland, 1997. Licenciate's thesis.

[38] P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, 1999. LNAI 1730.

[39] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.

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

[41] 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.

[42] M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today - Recent Trends and Developments*, pages 471–485. Springer-Verlag, 1995. LNCS 1000.

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

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A43	Kari J. Nurmela Minimum-Energy Point Charge Configurations on a Circular Disk. February 1997.
HUT-TCS-A44	Karsten Schmidt Applying Reduction Rules to Algebraic Petri nets. March 1997.
HUT-TCS-A45	Keijo Heljanko Model Checking the Branching Time Temporal Logic CTL. May 1997.
HUT-TCS-A46	Tuomas Aura Stateless connections. May 1997.
HUT-TCS-A47	Patrik Simons Towards Constraint Satisfaction through Logic Programs and the Stable Model Semantics. August 1997.
HUT-TCS-A48	Tuomas Aura On the structure of delegation networks. December 1997.
HUT-TCS-A49	Tomi Janhunen Non-Monotonic Systems: A Framework for Analyzing Semantics and Structural Properties of NMR. March 1998.
HUT-TCS-A50	Ilkka Niemelä (Ed.) Proceedings of the HeCSE Workshop on Emerging Technologies in Distributed Systems. March 1998.
HUT-TCS-A51	Kimmo Varpaaniemi On the Stubborn Set Method in Reduced State Space Generation. May 1998.
HUT-TCS-A52	Ilkka Niemelä, Torsten Schaub (Eds.) Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning. May 1998.
HUT-TCS-A53	Stefan Rönn Semantics of Semaphores. 1998.
HUT-TCS-A54	Antti Huima Analysis of Cryptographic Protocols via Symbolic State Space Enumeration. August 1999.
HUT-TCS-A55	Tommi Syrjänen A Rule-Based Formal Model For Software Configuration. December 1999.
HUT-TCS-A56	Keijo Heljanko Deadlock and Reachability Checking with Finite Complete Prefixes. December 1999.

ISBN 951-22-4884-0
ISSN 0783-5396