Deadlock Checking for Complete Finite Prefixes Using Logic Programs with Stable Model Semantics (Extended Abstract)

Keijo Heljanko

Helsinki University of Technology, Digital Systems Laboratory
P.O.Box 1100, FIN-02015 HUT, Finland
Keijo.Heljanko@hut.fi

Abstract. McMillan has presented a deadlock detection method based on complete finite prefixes (i.e. net unfoldings) of a Petri net. The problem of checking deadlock-freedom is NP-complete in the size of the prefix. McMillan originally suggested a branch-and-bound algorithm for deadlock detection in prefixes. Recently, Melzer and Römer have presented another algorithm which is based on solving mixed integer programming problems. We show that instead of using mixed integer programming, a constraint-based logic programming framework can be employed, and present a simple linear-size translation from deadlock detection in prefixes into the problem of finding a stable model of a logic program. We present experimental results from a straightforward prototype implementation combining the prefix generator of the PEP-tool, the translation, and an implementation of constraint-based logic programing framework, the `smodels` system. We find our approach competitive with the previous approaches.

1 Introduction

Petri nets are a model of concurrency which can be used to analyze e.g. reactive systems. One of the analysis problems associated with reactive systems is that of deadlock-freedom: Do all reachable markings enable some transition? For 1-safe Petri nets this problem is PSPACE-complete in the size of the net [2], however, restricted subclasses of 1-safe Petri nets exist for which this problem is NP-complete [7,8]. McMillan has presented a deadlock detection method based on complete finite prefixes (i.e. net unfoldings) of a Petri net [7,8]. 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 complete finite prefix, which is a 1-safe Petri net of a restricted form. The blowup of the translation depends on the problem instance, and experimental results show that it can in many cases be avoided [3,7–9].

In this work we will mainly discuss ways of solving the NP-complete deadlock detection problem for prefixes. McMillan originally suggested a branch-and-bound algorithm for deadlock detection in prefixes. Recently, Melzer and Römer have presented another algorithm which is based on solving mixed integer programming problems generated from prefixes [9]. We suggest that by using a third alternative, a constraint-based logic programming framework [10–12], this problem can be quite elegantly and efficiently solved.

First we present Petri net notations used in the paper. In Section 3 we will introduce the rule-based constraint programming framework. Section 4 contains the main result of this work, a simple linear-size translation from deadlock detection for prefixes into the problem of finding a stable model of a logic program. In Section 5 we present experimental results from a straightforward prototype implementation. In Section 6 we conclude and discuss directions for future research.

2 Petri Net Definitions

This section is included for completeness, it is based on the notation of [3,9].

2.1 Petri Nets

A triple $\langle S, T, F\rangle$ is a *net* if $S \cap T = \emptyset$ and $F \subseteq \langle S \times T\rangle \cup \langle T \times S\rangle$. 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 $\langle S \times T\rangle \cup \langle T \times S\rangle$. The *preset* of a node x , denoted by ${}^\bullet x$, is the set $\{y \in S \cup T | F(y, x) = 1\}$. The *postset* of a node x , denoted by $x^\bullet$, is the set $\{y \in S \cup T | F(x, y) = 1\}$. 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 multiset 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_n is *reachable* in Σ if there exist 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 it's reachable marking are 1-safe. In this work we will restrict ourselves to the set of 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 a nonempty pre- and postsets.

2.2 Occurrence Nets

We use $\leq_F$ to denote the reflexive transitive closure of F. Let $\langle S, T, F\rangle$ 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 $t_2 \leq_F x_2$. 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. A *configuration* C of an occurrence net is a set of events satisfying:

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

2.3 Branching Processes

Branching processes are "unfoldings" of net systems and were introduced by Engelfriet [1]. 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* 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 $\beta = \langle N', p \rangle$, where N' is a occurrence net, and p is a homomorphism from N' to $\langle S, T, F \rangle$ such that: the restriction of p to $Min(N')$ is a bijection between $Min(N')$ and M_0, and $\forall e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2 \wedge p(e_1) = p(e_2)$ then $e_1 = e_2$. The set of places associated with a configuration C of β is denoted by $Mark(C) = p((Min(N) \cup C^\bullet) \setminus {}^\bullet C)$.

2.4 Complete Finite Prefixes

A finite branching process β is a *complete finite prefix* of a net system Σ if and only 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 $p(e) = t$.

Algorithms to obtain a complete finite prefix β given a 1-safe net system Σ are presented in e.g. [3,7,8]. The algorithms will mark some events of the prefix β as special *cut-off events*, which we denote by the set *CutOffs*(β). The intuition behind cutoff events is that for each cutoff event e there already exists another event e' in the prefix. The markings after executing e can also be reached after executing e', and thus the markings after e need not to be considered any further. Due to space limitations we direct the reader interested in the approach to [3,7–9].

3 Rule-Based Constraint Programming

We will use normal logic programs with stable model semantics [4] as the underlying formalism into which the deadlock detection problem for complete finite prefixes is translated. This section is to a large extent based on [12].

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* [10], 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

$$h \leftarrow a_1, \ldots, a_n, \text{not } (b_1), \ldots, \text{not } (b_m) \tag{1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ and h are propositional atoms. Such a rule can be seen as a constraint saying that if atoms $a_1, \ldots, a_n$ are in a model and atoms $b_1, \ldots, b_m$ are not in a model, then atom h is in a 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 behaviour of the semantics using programs P1-P4:

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

Program P1 has two stable models: $\{a\}$ and $\{b\}$. 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 not (b) in order to deduce the stable model $\{a\}$. Program P2 has only the empty set as its only 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 not (a), then we will deduce a, which will contradict with our assumption not (a). Program P4 has one stable model $\{b\}$. If we assume not (a) then we will deduce b. If we assume not (b) then we can't deduce a, because c can't be deduced from our assumptions.

The stable model semantics for a normal logic program P is defined as follows [4]. 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 not (x) in its body such that $x \in A$ 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.

One interesting property of the stable model semantics is that only the atoms occurring as not-atoms in some program rule contribute to the search space. Therefore a non-deterministic way of constructing stable models is to guess which assumptions (not-atoms of the program) to use, and then check using deductive closure (in linear time) whether the resulting model agrees with the assumptions. The problem of determining the existence of a stable model is infact NP-complete [6].

3.1 The tool `smodels`

There is a tool, the `smodels` system [11,12], which provides an implementation of logic programs as a rule-based constraint programming framework. It has been developed to find (some or all) stable models of a logic program. It can also tell when the program has no stable models. It contains strong pruning techniques to make the problem tractable for a large class of programs. The `smodels` implementation needs only space linear in the size of the input program [12]. The stable model semantics is defined using rules of the form (1). As of version 2.0, `smodels` also handles an extended set of rules. All of these new rules can be seen as succinct encodings of sets of basic rules. In this work we will only need the extended rule of the form: $h \leftarrow 2\{a_1, \ldots, a_n\}$. The semantics of this rule is that if two or more atoms from the set $a_1, \ldots, a_n$ belong to the model, then also the atom 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: $h \leftarrow a_i, a_j$. Using an extended rule instead of the corresponding basic rule encoding was necessary to achieve an efficient translation of the problem at hand.

4 Translating Deadlock Checking into Logic Programs

In this section we present the translation of deadlock detection into logic programs. The main result can be seen as a rephrasing of the Theorem 4 of [9], where mixed integer programming has been replaced by the rule-based constraint programming framework. First we define some additional notation.

Definition 1. The set of non-cutoff events corresponding to the prefix $\beta = \langle N, h\rangle$ with $N = \langle B, E, F\rangle$ is $\mathit{NonCutOffs}(\beta) = \{e | e \in E \wedge e \notin \mathit{CutOffs}(\beta)\}$.

Definition 2. The set of normal events corresponding to the prefix $\beta = \langle N, h\rangle$ with $N = \langle B, E, F\rangle$ is $\mathit{NormalEvents}(\beta) = \mathit{NonCutOffs}(\beta) \setminus \{e | e \in E \wedge {}^\bullet e = \emptyset\}$.

Normal events include all non-cutoff events except the minimal elements, if such elements exist.

Next we present the main result of this work. We will discuss the theorem in full detail in the text following the theorem.

Theorem 3. *Let $\beta = \langle N, h\rangle$ with $N = \langle B, E, F\rangle$ be a complete finite prefix of a given n-safe net system Σ. For technical reasons we use a slightly modified prefix β', which is identical to β except that we have replaced the net N with the net $N' = \langle B', E', F'\rangle$, where $B' = B$, $E' = E \cup \{e_0\}$, $F' = F \cup \{\langle e_0, b\rangle | b \in Min(N)\}$. The event e_0 is a new minimal event which generates the initial marking.*

Σ is deadlock-free if and only if the logic program containing the following rules has no stable model:

1. *A rule:*
   $e_0 \leftarrow$
2. *For all $e_i \in \mathit{NormalEvents}(\beta')$ a rule:*
   $e_i \leftarrow e'_1, \ldots, e'_n, \text{not } (be_i)$,
   such that $\bigcup_{1 \le j \le n} \{e'_j\} = {}^\bullet({}^\bullet e_i)$
3. *For all $e_i \in \mathit{NormalEvents}(\beta')$ a rule:*
   $be_i \leftarrow \text{not } (e_i)$
4. *For all $b_i \in B'$ such that $|b_i^\bullet \setminus \mathit{CutOffs}(\beta')| \ge 1$ a rule:*
   $\mathit{conflict} \leftarrow 2\{e'_1, \ldots, e'_n\}$,
   such that $\bigcup_{1 \le j \le n} \{e'_j\} = b_i^\bullet \setminus \mathit{CutOffs}(\beta')$

5. *A rule:*
   $bottom \leftarrow \text{not } (bottom),\ conflict$

6. *For all* $b_i \in \{b \in B' | b^\bullet \neq \emptyset\}$ *a rule:*
   $b_i \leftarrow e, \text{not } (e'_1), \ldots, \text{not } (e'_n)$,
   such that $\{e\} = {}^\bullet b_i$, *and* $\bigcup_{1 \leq j \leq n} \{e'_j\} = b_i^\bullet \setminus CutOffs(\beta')$

7. *For all* $e_i \in NormalEvents(\beta') \cup CutOffs(\beta')$ *a rule:*
   $live \leftarrow b_1, \ldots, b_n$,
   such that $\bigcup_{1 \leq j \leq n} \{b_j\} = {}^\bullet e_i$

8. *A rule:*
   $bottom \leftarrow \text{not } (bottom),\ live$

The intuition behind the logic program is the following: Rules 1-5 are a redefinition of a legal configuration in terms of logic programs. The stable models of the program containing only rules 1-5 have a one-to-one correspondence with those configurations of β which do not include any cut-off events. Rules 6-8 are additional constraints to these configurations. They remove from this set of configurations all such configurations in which any event of the prefix is enabled. The remaining configurations (if any exist) are configurations in which no event is enabled i.e. deadlock configurations.

We'll now discuss the program in more detail. The program has the atoms:

- The atom e_i is in a model when the event e_i is in the set of fired events.
- The atom be_i is in a model when the event e_i is not in the set of fired events.
- The atom *conflict* is deduced when two or more events sharing preset conditions are in the set of fired events, and thus the set of fired events is not a configuration.
- The atom *bottom* is used merely for technical reasons. It is used to exclude all stable models containing either atom *conflict* or *live*.
- The atom b_i is in a model when the condition b_i holds a token after the set of fired events.
- The atom *live* is deduced when any net event is enabled.

The program rules do the following:

1. Rule 1 establishes the initial marking of the net by requiring that the event generating the initial marking is always in the set of fired events. The configuration containing only the event e_0 in the net N' thus corresponds to the empty configuration in the net N.
2. Rule 2 says that an event e_i is in the set of fired events, if all of the events which generate it's preset are in the set of fired events, and the atom be_i is not in the model.
3. Rule 3 says that atom be_i is in a model if event e_i is not in the set of fired events. This is a technicality which makes it legal for an event to be enabled and not to be necessarily fired. This is needed to make also non-maximal configurations to have stable models.

4. Rule 4 says that if two or more events which share a preset place are in the set of fired events, then the atom *conflict* will be in the model.
5. Rule 5 excludes all models containing the atom *conflict*. Therefore rules 4 and 5 together disallow sets of fired events containing immediate conflicts due to shared presets.
6. Rule 6 makes a condition to be in a model when its preset event, and none of its postset events are in the set of fired events.
7. Rule 7 says that an event is enabled if all its preset conditions are in the model. Note that this also includes cut-off events.
8. Rule 8 excludes all models in which any of the events are enabled, i.e. the atom *live* is in the model.

It is easy to see that the size of translated program is linear in the size of the prefix i.e. $O(|B|+|E|+|F|)$. Because the rule-based constraint programming system only needs linear space in the size of the input program, deadlock checking exploiting this translation can be made using only linear space in the size of the prefix. The translation is also local, which makes it is quite straightforward to implement the translation in linear time in the size of the prefix.

5 Prototype Implementation

We have implemented the translation described in the previous section using the interpreted scripting language Python. It translates the deadlock checking for complete finite prefix generated by the PEP-tool [5] into a logic program. The only optimization the translator script does is that it removes duplicate rules, which can be done in polynomial time. (Duplicate rules might arise from rules 4 and 7.) The Python script `gen` inputs an ASCII file which describes the complete finite prefix, and generates another ASCII file which contains the logic program. This ASCII file is then parsed by the smodels parser `pparse` into internal form suitable for the `smodels` stable model generator. This prototype implementation was created to research the feasibility of the approach, rather than to be a fully functional tool. The `gen` script, the `pparse` program, and the `smodels` computational engine will eventually be all integrated into one tool which directly reads binary format prefix files. Initial results show that this will eliminate almost all of the costs associated with the generation and parsing steps.

5.1 Experimental Results

We have made experiments with our approach using the examples by Corbett and McMillan, which were used by Melzer and Römer in [9]. The Figures 1 and 2 present the running times in seconds for the various algorithms used in this work and also those presented in [9]. The running times have been measured using a Pentium 166MHz, 64MB RAM, 128MB swap, Linux 2.0.29, g++ 2.7.2.1, pparse 1.4, smodels pre-2.0.11, and PEP 1.6g. The running times for the experiments by Melzer and Römer were conducted on a Sparcstation 20/712, 96MB RAM.

The rows of the table correspond to different problems. The columns represent: sum of user and system times measured by `/usr/bin/time` command, or times reported in [9], depending on the column:

– Unf[1] = time for unfolding (PEP)
– Gen[1] = time for constraint program generation (`gen`)
– Parse[1] = time for parsing the constraint program (`pparse`)
– DC_MIP[2] = time for Mixed integer programming algorithm in [9]
– DC_McM[2] = time for McMillan's algorithm in [9]
– DC_McM[1] = time for McMillan's algorithm using Pentium 166MHz
– DC_smo[1] = time for `smodels` to determine whether there is a deadlock

A marking $mem(n)$ notes that the program ran out of memory after n seconds [9]. The marking $vm(n)$ notes that the program ran out of virtual memory after n seconds.

Problem(size)	Unf[1]	Gen[1]	Parse[1]	DC_MIP[2]	DC_McM[2]	DC_McM[1]	DC_smo[1]
DPD(4)	0.13	6.2	0.2	2.0	0.3	0.2	0.3
DPD(5)	0.58	16.3	0.5	17.3	1.9	1.7	1.2
DPD(6)	3.20	39.1	1.2	82.8	20.2	12.1	4.9
DPD(7)	17.43	91.0	2.8	652.6	234.0	129.1	16.0
DPH(4)	0.13	6.9	0.2	1.8	0.3	0.2	0.3
DPH(5)	1.26	27.3	0.7	42.9	10.5	6.4	2.3
DPH(6)	33.90	150.1	3.9	1472.8	1907.6	1100.0	29.9
DPH(7)	934.03	931.5	23.0	-	-	$vm(1713.3)$	606.0
ELEVATOR(1)	0.09	3.3	0.1	0.1	0.0	0.1	0.1
ELEVATOR(2)	0.54	16.0	0.4	2.3	0.9	0.5	2.8
ELEVATOR(3)	10.22	78.0	1.9	14.5	18.7	10.1	63.4
ELEVATOR(4)	188.17	368.6	9.2	387.8	492.7	269.0	1221.7
FURNACE(1)	0.10	5.6	0.1	0.3	0.2	0.2	0.1
FURNACE(2)	3.25	52.3	1.1	18.1	19.0	10.7	2.3
FURNACE(3)	136.88	358.2	8.9	1112.5	$mem(811.1)$	$vm(387.6)$	57.6
RING(3)	0.03	1.2	0.1	0.1	0.0	0.0	0.1
RING(5)	0.08	3.6	0.1	1.3	0.1	0.1	0.3
RING(7)	0.24	8.4	0.3	17.1	0.3	0.2	1.6
RING(9)	0.73	16.6	0.5	71.2	1.1	0.7	7.3
RW(6)	0.09	7.9	0.1	0.7	0.5	0.3	0.1
RW(9)	2.06	108.3	0.9	58.5	122.3	69.8	0.6
RW(12)	138.89	2682.9	10.0	24599.9	$mem(6004.9)$	$vm(3111.8)$	5.3

Fig. 1. Measured running times in seconds:
[1] = Pentium 166MHz, 64MB RAM, Linux 2.0.29.
[2] = Sparcstation 20/712, 96MB RAM [9]

It is difficult comment on the absolute running times of algorithms running on different machines. Some remarks on the scalability of the results inside the problem instances can however be made. Our approach is scaling better than either of the other methods on the problems DPD, DPH, FURNACE, RW, DME, and SYNC. On the other hand, it seems to be doing worse than McMillan's algorithm on RING. The scaling between ELEVATOR(3) and ELEVATOR(4) is better with our approach than either of the two other algorithms, but larger instances would be needed to draw any conclusions about this.

Problem(size)	Unf[1]	Gen[1]	Parse[1]	DC_MIP[2]	DC_McM[2]	DC_McM[1]	DC_smo[1]
DME(2)	0.13	4.6	0.2	1.9	0.07	0.07	0.32
DME(3)	0.36	11.5	0.3	64.6	0.50	0.35	2.34
DME(4)	1.09	23.1	0.6	216.1	1.67	1.41	9.69
DME(5)	3.19	40.4	1.1	1968.3	7.83	5.60	31.62
DME(6)	8.23	63.7	1.7	13678.3	26.43	21.42	87.18
DME(7)	18.21	96.5	2.6	-	97.80	67.84	204.98
DME(8)	37.56	140.4	3.7	-	251.52	184.51	425.96
DME(9)	70.44	197.3	5.1	-	701.74	527.02	823.78
DME(10)	124.20	270.5	6.9	-	1801.48	1273.94	1483.96
DME(11)	207.64	366.6	9.0	-	4682.36	2892.92	2541.65
SYNC(2)	4.61	42.0	1.4	171.6	69.0	36.9	21.91
SYNC(3)	219.43	322.8	9.8	11985.0	26621.7	14219.0	626.16

Fig. 2. Measured running times in seconds:
[1] = Pentium 166MHz, 64MB RAM, Linux 2.0.29.
[2] = Sparcstation 20/712, 96MB RAM [9]

The ELEVATOR is the only problem which contains a deadlock, and this might make it behave differently from the other problems. It is also the only example in which the `smodels` computation engine had to make one choice of firing a transition to find the deadlock. On all the other examples the strong pruning techniques of `smodels` implementation removed all of the search space, thus having a guaranteed polynomial running time in the size of the problem instance. We need to have a larger set of examples in the future containing also difficult cases, which must exist due to the complexity of the problem.

6 Conclusions

We have presented using a constraint-based logic programming framework for detecting deadlocks from complete finite prefixes. Our main result is a simple linear-size translation from deadlock detection on prefixes into the problem of finding a stable model of a normal logic program. We present experimental results from a straightforward prototype implementation, and find our approach competitive with the previous approaches.

For future work we will extend this approach to the class of reachability problems on prefixes. We conjecture that the translation can also be extended to the class of Petri nets which have the following properties: 1-safe, acyclic, and each transition can occur at most once. We will also need a larger set of examples to evaluate the approach against other approaches, and also to test the the robustness of the various algorithms to changes in the input representation. Also a more optimized translation from the prefixes into rule-based constraint programs needs to be implemented. Even by using only linear time in the size of the prefix a lot of optimizations exploiting the structure of the prefix can be made. If needed, runtime overhead can also be reduced by creating a special purpose tool which integrates all the phases of the translation, and additionally might use a search algorithm which only handles the very restricted set of rule-based constraint programs created by the translation.

7 Acknowledgements

The author would like to thank Ilkka Niemelä for introducing him into the rule-based constraint programming framework, and for many constructive ideas for the translation. The tool `smodels` was programmed by Patrik Simons, who also gave valuable support for its usage. Stephan Melzer and Stefan Römer provided the example nets they used, and also Linux binaries for the McMillan's algorithm, which both were invaluable for conducting the experiments. The financial support of Helsinki Graduate School on Computer Science and Engineering (HeCSE), and the Academy of Finland are gratefully acknowledged.

References

1. J. Engelfriet. Branching processes of Petri nets. In *Acta Informatica 28*, pages 575–591, 1991.
2. J. Esparza and M. Nielsen. Decidability issues for Petri Nets - a survey. *Journal of Information Processing and Cybernetics 30(3)*, pages 143–160, 1994.
3. 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.
4. 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, August 1988. The MIT Press.
5. B. Grahlmann. The PEP Tool. In *Proceedings of CAV'97 (Computer Aided Verification)*, pages 440–443. Springer-Verlag, June 1997. LNCS 1254.
6. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.
7. 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.
8. K. L. McMillan. A technique of a state space search based on unfolding. In *Formal Methods is System Design 6(1)*, pages 45–65, 1995.
9. 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.
10. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, Trento, Italy, May 1998. Helsinki University of Technology, Digital Systems Laboratory, Research Report A52.
11. 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, July 1997. Springer-Verlag.
12. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report A47, Helsinki University of Technology, Espoo, Finland, August 1997. Licenciate's thesis, Available at http://saturn.hut.fi/pub/reports/A47.ps.gz.