Bounded LTL Model Checking with Stable Models [⋆]

Keijo Heljanko and Ilkka Niemelä

Helsinki University of Technology
Dept. of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
{Keijo.Heljanko, Ilkka.Niemela}@hut.fi

Abstract. In this paper bounded model checking of asynchronous concurrent systems is introduced as a promising application area for answer set programming. As the model of asynchronous systems a generalization of communicating automata, 1-safe Petri nets, are used. It is shown how a 1-safe Petri net and a requirement on the behavior of the net can be translated into a logic program such that the bounded model checking problem for the net can be solved by computing stable models of the corresponding program. The use of the stable model semantics leads to compact encodings of bounded reachability and deadlock detection tasks as well as the more general problem of bounded model checking of linear temporal logic. Some experimental results on solving deadlock detection problems using the translation and the `Smodels` system are presented.

1 Introduction

In this paper we put forward symbolic model checking [2, 3] as a promising application area for answer set programming systems. In particular, we demonstrate how bounded model checking problems of asynchronous concurrent systems can be reduced to computing stable models of logic programs.

Verification of asynchronous systems is typically done by enumerating the set of reachable states of the system. Tools based on this approach (with various enhancements) include, e.g., the Spin system [12], which supports extended state machines communicating through FIFO queues, and the PROD tool [17] based on Petri nets. The main problem with enumerative model checkers is the amount of memory needed to store the set of reachable states.

Symbolic model checking is widely applied especially in hardware verification. The main analysis technique is based on (ordered) binary decision diagrams (BDDs). In many cases the set of reachable states can be represented very compactly using a BDD encoding. Although the approach has been successful, there

[⋆] This is an extended version of a paper titled "Answer Set Programming and Bounded Model Checking"[11] presented at the AAAI Spring 2001 Symposium on Answer Set Programming, Stanford, March 2001. The financial support of Academy of Finland (Projects 43963, 47754) and Tekniikan Edistämissäätiö are gratefully acknowledged.

are difficulties in applying BDD-based techniques, in particular, in areas outside hardware verification. The key problem is that some Boolean functions do not have a compact representation as BDDs and the size of the BDD representation of a Boolean function is very sensitive to the variable ordering used. Bounded model checking [1] has been proposed as a technique for overcoming the space problem by replacing BDDs with satisfiability (SAT) checking techniques because typical SAT checkers use only polynomial amount of memory. The idea is roughly the following. Given a sequential digital circuit, a (temporal) property to be verified, and a bound n, the behavior of a sequential circuit is unfolded up to n steps as a Boolean formula S and the negation of the property to be verified is represented as a Boolean formula $\overline{R}$. The translation to Boolean formulae is done so that $S \wedge \overline{R}$ is satisfiable iff the system has a behavior violating the property of length at most n. Hence, bounded model checking provides directly interesting and practically relevant benchmarks for any answer set programming system capable of handling propositional satisfiability problems.

Until now bounded model checking has been applied to synchronous hardware verification and little attention has been given to knowledge representation issues such as developing concise and efficient logical representation of system behavior. In this work we study the knowledge representation problem and employ ideas used in reducing planning to stable model computation [15]. The aim is to develop techniques such that the behavior of an asynchronous concurrent system can be encoded compactly and the inherent concurrency in the system could be exploited in model checking the system. To illustrate the approach we use a simple basic Petri net model of asynchronous systems, 1-safe Place/Transition nets, which is an interesting generalization of communicating automata [5].

The structure of the rest of the paper is the following. In the next section we introduce Petri nets and the bounded model checking problem. Then we develop a compact encoding of bounded model checking as the problem of finding stable models of logic programs. We first show how to treat reachability properties such as deadlocks and then demonstrate how to extend the approach to cope with properties expressed in linear temporal logic (LTL). We discuss initial experimental results and end with some concluding remarks.

2 Petri nets and bounded model checking

We will now introduce P/T-nets. They are one of the simplest forms of Petri nets. We will use as a running example the P/T-net presented in Fig. 1.

A triple $\langle P, T, F \rangle$ is a *net* if $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$. The elements of P are called *places*, and the elements of T *transitions*. Places and transitions are also called *nodes*. The places are represented in graphical notation by circles, transitions by squares, and the *flow relation* F with arcs. We identify F with its characteristic function on the set $(P \times T) \cup (T \times P)$. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. In our running example, e.g., ${}^\bullet t2 = \{p1, p2\}$. The *postset* of a node x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. Again in our running example $p2^\bullet = \{t2, t3, t5\}$.

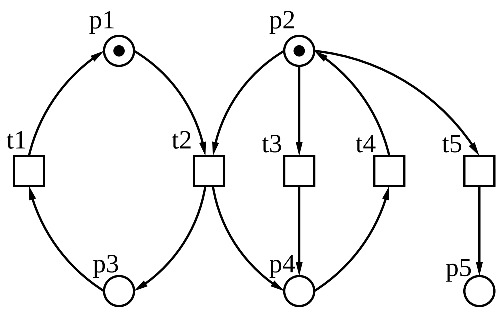

Fig. 1. Running Example

A *marking* of a net $\langle P, T, F \rangle$ is a mapping $P \mapsto \mathbb{N}$. A marking M is identified with the multi-set which contains $M(p)$ copies of p for every $p \in P$. A 4-tuple $\Sigma = \langle P, T, F, M_0 \rangle$ is a *net system* (also called a *P/T-net*) if $\langle P, T, F \rangle$ is a net and M_0 is a marking of $\langle P, T, F \rangle$. A marking is graphically denoted by a distribution of tokens on the places of the net. In our running example in Fig. 1 the net has the initial marking $M_0 = \{p1, p2\}$.

A marking M enables a transition $t \in T$ if $\forall p \in P : F(p,t) \leq M(p)$. If t is enabled, it can *occur* leading to a new marking (denoted $M \xrightarrow{t} M'$), where M' is defined by $\forall p \in P : M'(p) = M(p) - F(p,t) + F(t,p)$. In the running example $t2$ is enabled in the initial marking M_0, and thus $M_0 \xrightarrow{t2} M'$, where $M' = \{p3, p4\}$.

A marking M_n is *reachable* in Σ if there is an *execution*, i.e., a (possibly empty) 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 marking M is reachable within a bound n, if there is an execution with $\leq n$ transitions, with which M is reachable.

A marking M is 1-safe if $\forall p \in P : M(p) \leq 1$. A P/T-net is 1-safe if all its reachable markings are 1-safe. We will restrict ourselves to finite P/T-nets which are 1-safe, and in which each transition has both nonempty pre- and postsets.

Given a 1-safe P/T-net Σ, we say that a set of transitions $S \subseteq T$ is *concurrently enabled* in the marking M, if (i) all transitions $t \in S$ are enabled in M, and (ii) for all pairs of transitions $t, t' \in S$, such that $t \neq t'$, it holds that ${}^\bullet t \cap {}^\bullet t' = \emptyset$. If a set S is concurrently enabled in the marking M, we can fire it in a *step* (denoted $M \xrightarrow{S} M'$), where M' is the marking reached after firing all of the transitions in the step S in arbitrary order. It is easy to prove by using the 1-safeness of the P/T-net Σ that all possible interleavings of transitions in a step S are enabled in M, and that they all lead to the same final marking M'. In our running example in the marking $M' = \{p3, p4\}$ the step $\{t1, t4\}$ is enabled, and will lead back to the initial marking M_0. This is denoted by $M' \xrightarrow{\{t1,t4\}} M_0$. Notice also that for any enabled transition, the singleton set containing only that transition is always (trivially) a step.

We say that a marking M_n is *reachable in step semantics* in a 1-safe P/T-net if there is a *step execution*, i.e., a (possibly empty) sequences $S_1, S_2, \ldots, S_n$ of steps and $M_1, M_2, \ldots, M_{n-1}$ of markings such that: $M_0 \overset{S_1}{\to} M_1 \overset{S_2}{\to} \ldots M_{n-1} \overset{S_n}{\to} M_n$. A marking M is reachable within a bound n in the step semantics, if there is a step execution with at most n steps, with which M is reachable.

We will refer to the "normal semantics" as *interleaving semantics*. Note that if a marking is reachable in n transitions in the interleaving semantics, it is also reachable in n steps in the step semantics. However, the converse does not necessarily hold. We have, however, the following theorem.

Theorem 1. *For finite 1-safe P/T-nets the set of reachable markings in the interleaving and step semantics coincide.*

Linear temporal logic (LTL). The linear temporal logic LTL is one of the most widely used logic for specifying properties of reactive systems [3]. The basic idea is to specify properties that the system should have using LTL. A model checker is then used to check whether all (infinite) behaviors of the system are models of the specification formula. If not, then the model checker outputs a behavior of the system which violates the given specification.

Given a finite set AP of atomic propositions, the syntax of LTL[1] is given by:

$$\varphi ::= p \in AP \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, U \, \varphi_2 \mid \varphi_1 \, R \, \varphi_2 \ .$$

An ω-word over 2^{AP} is an infinite sequence $w = x_0 \, x_1 \, \ldots$ such that $x_i \in 2^{AP}$ for all $i \geq 0$. For an ω-word w we define $w_{(i)} = x_i$, and denote by $w^{(i)}$ the suffix of w starting at x_i. We define the relation $w \models \varphi$ inductively as follows:

- $w \models p$ iff $p \in w_{(0)}$ for $p \in AP$
- $w \models \neg\varphi_1$ iff not $w \models \varphi_1$
- $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$
- $w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$
- $w \models \varphi_1 \, U \, \varphi_2$ iff there exists a $j \geq 0$ such that $w^{(j)} \models \varphi_2$ and for all $0 \leq i < j$, $w^{(i)} \models \varphi_1$
- $w \models \varphi_1 \, R \, \varphi_2$ iff for all $j \geq 0$, if for every $i < j$ $w^{(i)} \not\models \varphi_1$ then $w^{(j)} \models \varphi_2$.

We define some shorthand LTL formulas: $\top \equiv p \vee \neg p$ for some arbitrary fixed $p \in AP$, $\bot \equiv \neg\top$, $\Diamond \varphi \equiv (\top \, U \, \varphi)$, $\Box \varphi \equiv (\bot \, R \, \varphi)$, and $\varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$.

The temporal operators are called: U for "until", R for "release", $\Diamond$ for "eventually", and $\Box$ for "globally". Some examples of practical use of LTL formulas in specification are: $\Box\neg(cs_1 \wedge cs_2)$ (it always holds that two processes are not at the same time in a critical section), $\Box(req \to \Diamond ack)$ (it is always the case that a request is eventually followed by an acknowledgement), and $((\Box\Diamond sch_1) \wedge (\Box\Diamond sch_2)) \to (\Box(tr_1 \to \Diamond cs_1))$ (if both process 1 and 2 are scheduled infinitely often, then always the entering of process 1 in the trying section is followed by the process 1 eventually entering the critical section).

[1] Note that we do not define the often used next-time operator $X \, \varphi$. This is a tradeoff which allows the use of step semantics.

Given a 1-safe P/T net Σ, we use a chosen subset of the places as the atomic propositions AP. An infinite (interleaving) execution $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ satisfies φ iff the corresponding ω-word $w = (M_0 \cap AP), (M_1 \cap AP), \ldots$ satisfies φ. We say that Σ satisfies φ iff every infinite execution starting from the initial marking M_0 satisfies φ. Alternatively, Σ does not satisfy φ if there exists an infinite execution starting from M_0 which satisfies $\neg\varphi$. We call such an execution a *counterexample.*

The temporal logic LTL specifies properties of infinite executions. In many cases it suffices to reason about simple temporal properties. A typical example is the reachability of a marking satisfying some condition C which roughly corresponds to finding a counterexample for a formula $\Box\neg C$. An important reachability based property is deadlock detection.

Definition 1. (Deadlock) *Given a 1-safe P/T-net Σ, is there a reachable marking M which does not enable any transition of Σ?*

Most analysis questions including deadlock detection and LTL model checking are PSPACE-complete in the size of a 1-safe Petri net, see e.g., [6]. In *bounded model checking* we fix a bound n and look for counterexamples which are shorter than the given bound n. For example, in the case of *bounded deadlock detection* in step semantics we look for step executions reaching a deadlock in n steps. It is easy to show that, e.g., the bounded deadlock detection problem in step semantics is NP-complete (when the bound n is given in unary coding).

This idea can also be applied to LTL model checking. Biere et.al. [1] introduce *bounded LTL model checking.* They also discuss how to ensure that a given bound n is sufficient to guarantee completeness. Unfortunately, getting an exact bound is often computationally infeasible, and easily obtainable upper bounds are too large. In the case of 1-safe P/T-nets they are exponential in the number of places in the net. Therefore the bounded model checking results are usually not conclusive if a counterexample is not found. Thus bounded model checking is at its best in "bug hunting", and not as easily applicable in verifying systems to be correct.

3 From bounded model checking to answer set programming

In this section we show how to solve bounded LTL model checking problems using answer set programming. We start with the simpler reachability properties and then extend the approach to handle full LTL model checking.

For encoding bounded model checking problems we use normal logic programs with the stable model semantics [8]. A normal rule is of the form

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \qquad (1)$$

where each a, b_i, c_j is a ground atom. We employ three extensions which can be seen as compact shorthands for normal rules. We use *integrity constraints*, i.e., rules with empty head. Such a constraint like the one on the left can be taken as a shorthand for a rule given on the right

$$\leftarrow b, \text{not } c \qquad \rightsquigarrow \qquad f \leftarrow \text{not } f, b, \text{not } c$$

where f is a new atom. For expressing the choice whether to include an atom in a stable model we use *choice rules*. They are normal rules where the head is in brackets with the idea that the head can be included in a stable model only if the body holds but it can be left out, too. Such a construct can be represented using normal rules by introducing a new atom. For example, the choice rule on the left corresponds to the two normal rules on the right where a' is a new atom.

$$\{a\} \leftarrow b, \text{not } c \qquad \rightsquigarrow \qquad \begin{array}{l} a \leftarrow \text{not } a', b, \text{not } c \\ a' \leftarrow \text{not } a \end{array}$$

Finally, a compact encoding of *conflicts* is needed, i.e., rules of the form

$$\leftarrow 2\{a_1, \ldots, a_n\} \tag{2}$$

saying that a stable model cannot contain any two atoms out of a set of atoms $\{a_1, \ldots, a_n\}$. Such a rule can be expressed, e.g., by adding a rule $f \leftarrow \text{not } f, a_i, a_j$ for each pair a_i, a_j from $\{a_1, \ldots, a_n\}$, i.e., using $\mathcal{O}(n^2)$ rules. Choice and conflict rules are simple cases of cardinality constraint rules [16]. The `Smodels` system (`http://www.tcs.hut.fi/Software/smodels/`) provides an implementation for cardinality constraint rules and includes primitives supporting directly such constraints without translating them first to corresponding normal rules.

3.1 Reachability checking

Now we devise a method for translating bounded reachability problems of 1-safe P/T-nets to tasks of finding stable models. Consider a net $N = \langle P, T, F \rangle$ and a step bound $n \geq 1$. We construct a logic program $\Pi_{\mathrm{A}}(N, n)$, which captures the possible executions of N up to n steps, as follows.

- For each place $p \in P$, include a choice rule $\{p(0)\} \leftarrow$.
- For each transition $t \in T$, and for all $i = 0, 1, \ldots, n-1$, include a rule

  $$\{t(i)\} \leftarrow p_1(i), \ldots, p_l(i) \tag{3}$$

  where $\{p_1, \ldots, p_l\}$ is the preset of t. Hence, a stable model can contain a transition instance in step i only if its preset holds at step i.
- For each place $p \in P$, for each transition t_k in the preset of p, and for all $i = 0, 1, \ldots, n-1$, include a rule

  $$p(i+1) \leftarrow t_k(i)\ . \tag{4}$$

  These say that p holds in the next step if at least one of its preset transitions is in the current step.

$$\begin{array}{llll}
\{t1(i)\} \leftarrow p3(i) & p1(i+1) \leftarrow t1(i) & p1(i+1) \leftarrow p1(i), \mathrm{not}\ t2(i) & \{p1(0)\} \leftarrow \\
\{t2(i)\} \leftarrow p1(i), p2(i) & p2(i+1) \leftarrow t4(i) & p2(i+1) \leftarrow p2(i), \mathrm{not}\ t2(i), & \{p2(0)\} \leftarrow \\
\{t3(i)\} \leftarrow p2(i) & p3(i+1) \leftarrow t2(i) & \qquad\qquad \mathrm{not}\ t3(i), \mathrm{not}\ t5(i) & \{p3(0)\} \leftarrow \\
\{t4(i)\} \leftarrow p4(i) & p4(i+1) \leftarrow t2(i) & p3(i+1) \leftarrow p3(i), \mathrm{not}\ t1(i) & \{p4(0)\} \leftarrow \\
\{t5(i)\} \leftarrow p2(i) & p4(i+1) \leftarrow t3(i) & p4(i+1) \leftarrow p4(i), \mathrm{not}\ t4(i) & \{p5(0)\} \leftarrow \\
 & p5(i+1) \leftarrow t5(i) & p5(i+1) \leftarrow p5(i) & \\
 & \multicolumn{2}{l}{\leftarrow 2\{t2(i), t3(i), t5(i)\} \ \text{ where } i = 0, 1, \ldots n-1} &
\end{array}$$

Fig. 2. Program $\Pi_{\mathrm{A}}(N, n)$

– For each place $p \in P$, and for all $i = 0, 1, \ldots, n-1$, include a rule

$$\leftarrow 2\{t_1(i), \ldots, t_l(i)\} \tag{5}$$

where $\{t_1, \ldots, t_l\}$ is the set of transitions having each p in their preset and $l \geq 2$. This rule states that at most one of the transitions that are in conflict w.r.t. p can occur.

– For each place p, and for all $i = 0, 1, \ldots, n-1$,

$$p(i+1) \leftarrow p(i), \mathrm{not}\ t_1(i), \ldots, \mathrm{not}\ t_l(i) \tag{6}$$

where $\{t_1, \ldots, t_l\}$ is the set of transitions having p in their preset. This is the *frame axiom* for p stating that p holds if no transition using it occurs.

Consider net N in Fig. 1 for which program $\Pi_{\mathrm{A}}(N, n)$ is given in Fig. 2. In $\Pi_{\mathrm{A}}(N, n)$ the initial marking is not constrained but any Boolean combination C of marking conditions can be captured with a set of rules $\Pi_{\mathrm{M}}(C, i)$ [16]. For example, to eliminate stable models not satisfying a condition C at step i saying that $M(p_1) = 1$ and ($M(p_2) = 0$ or $M(p_3) = 1$), it is sufficient to use rules $\Pi_{\mathrm{M}}(C, i)$:

$$\begin{array}{ll}
\leftarrow \mathrm{not}\ c(i) & c_{\bar{p_2} \vee p_3}(i) \leftarrow \mathrm{not}\ p_2(i) \\
c(i) \leftarrow p_1(i), c_{\bar{p_2} \vee p_3}(i) & c_{\bar{p_2} \vee p_3}(i) \leftarrow p_3(i)
\end{array}$$

Our approach can solve a reachability problem for a set of initial markings given by a condition C_0 where the markings to be reached are specified by another condition C.

Theorem 2. *Let $N = \langle P, T, F \rangle$ be a 1-safe P/T-net for all initial markings satisfying a condition C_0. Net N has an initial marking satisfying C_0 such that a marking satisfying a condition C is reachable in at most n steps iff $\Pi_{\mathrm{M}}(C_0, 0) \cup \Pi_{\mathrm{A}}(N, n) \cup \Pi_{\mathrm{M}}(C, n)$ has a stable model.*

The deadlock detection problem is now just a special case of a reachability property, just add rules $\Pi_{\mathrm{M}}(C, n) = \Pi_{\mathrm{D}}(N, n)$ eliminating stable models where some transition is enabled. Program $\Pi_{\mathrm{D}}(N, n)$ includes for each transition $t \in T$ and its preset $\{p_1, \ldots, p_l\}$, a rule

$$\leftarrow p_1(n), \ldots, p_l(n)\ . \tag{7}$$

For our running example, the rules $\Pi_{\mathrm{D}}(N, n)$ are

$$\leftarrow p3(n) \qquad \leftarrow p1(n), p2(n) \qquad \leftarrow p2(n) \qquad \leftarrow p4(n)\ .$$

3.2 Bounded LTL model checking

Our strategy for finding counterexamples for LTL formula φ (i.e., executions satisfying $\neg\varphi$) is exactly the same as in [1]. There it is shown to be an approximation of the unbounded version which becomes equivalent to the unbounded case if the bound used is sufficiently increased. We (as they do) require that all reachable states of the system have a successor (i.e., there are no deadlocks). In this case the reachability of a marking satisfying a condition C is equivalent to finding a counterexample for an LTL formula of the form $\Box\neg C$.

We look for two different kinds of counterexamples. On the left in Fig. 3 is a *loop counterexample*, and on the right is a *counterexample without loop.* Loop counterexamples specify an infinite execution themselves, while counterexamples without a loop specify a prefix of an execution, which can be always extended to an infinite execution (by the deadlock freeness assumption). The arcs of the figure denote the "next state" of each state. Notice in the loop counterexample that if $M_{(i-1)}$ is equivalent to the last state M_n, the state M_i is the "next state" of M_n. Our semantics is cautious in the case without loop, and extending the execution into an infinite one in any way will yield a counterexample.[2]

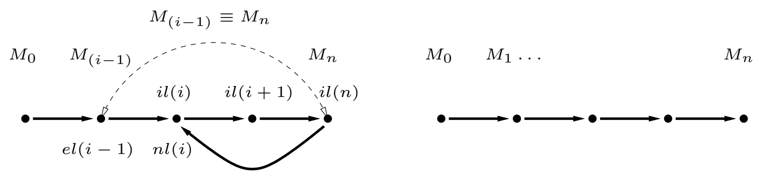

Fig. 3. Two counterexample possibilities

An LTL formula is said to be in *positive normal form* when all negations in the formula appear directly before an atomic proposition. A formula can be put into positive normal form with the following equivalences (and their duals): $\neg\neg\varphi \equiv \varphi$, $\neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2$, and $\neg(\varphi_1 \, U \, \varphi_2) \equiv \neg\varphi_1 \, R \, \neg\varphi_2$.

Given an LTL formula f in positive normal form (when the formula to be model checked is φ, the formula f is equivalent to $\neg\varphi$ with negations pushed in), and a bound $n \geq 1$ we construct a program $\Pi_{\mathrm{LTL}}(f, n)$ as follows.

- Guess which state is equivalent to the last. For all $0 \leq i \leq n-1$ add rule
$$\{el(i)\} \leftarrow \ . \tag{8}$$
- Disallow guessing two or more. (Guessing none is allowed though.) Add rule
$$\leftarrow 2\{el(0), el(1), \ldots, el(n-1)\} \ . \tag{9}$$

[2] Actually the counterexamples without loop are exactly the informative safety counterexamples of [13].

Formula type	Translation	Formula type	Translation
p, for $p \in AP$	$f(i) \leftarrow p(i)$	$\neg p$, for $p \in AP$	$f(i) \leftarrow \text{not } p(i)$
$f_1 \vee f_2$	$f(i) \leftarrow f_1(i)$ $f(i) \leftarrow f_2(i)$	$f_1 \wedge f_2$	$f(i) \leftarrow f_1(i), f_2(i)$
$f_1 \, U \, f_2$	$f(i) \leftarrow f_2(i)$ $f(i) \leftarrow f_1(i), f(i+1)$ $f(n+1) \leftarrow nl(i), f(i)$	$f_1 \, R \, f_2$	$f(i) \leftarrow f_2(i), f_1(i)$ $f(i) \leftarrow f_2(i), f(i+1)$ $f(n+1) \leftarrow nl(i), f(i)$ $f(n+1) \leftarrow l, \text{not } cstate(f)$ $cstate(f) \leftarrow il(i), \text{not } f_2(i)$

Fig. 4. Translation of an LTL formula f

– Check that the guess is correct. For all $0 \le i \le n-1$, $p \in P$ include rules

$$\leftarrow el(i), p(i), \text{not } p(n) \qquad \leftarrow el(i), p(n), \text{not } p(i) \; .$$

– Specify auxiliary loop related atoms. For all $0 \le i \le n-1$, include rules

$$l \leftarrow el(i) \qquad nl(i+1) \leftarrow el(i) \qquad il(i+1) \leftarrow el(i) \qquad il(i+1) \leftarrow il(i) \; .$$

See Fig. 3 for an example. The $nl(i)$ atom is in a model for the "next state" of the last state, while $il(i)$ is in the model for all states in the loop.

– Require that if a loop exists, the last step contains a transition to disallow looping by idling. Add the rule

$$\leftarrow l, \text{not } t_1(n-1), \ldots, \text{not } t_k(n-1) \tag{10}$$

where $\{t_1, \ldots, t_k\} = T$, i.e., the set of all transitions.

– Allow at most one visible transition in a step to eliminate steps which cannot be interleaved to yield a counterexample. For all $0 \le i \le n-1$, add rule

$$\leftarrow 2\{t_1(i), \ldots, t_k(i)\} \tag{11}$$

where $\{t_1, \ldots, t_k\}$ is the set of *visible transitions*, i.e., the transitions whose firing changes the marking of a place p appearing in the formula f.

We recursively translate the formula f by first translating its subformulae, and then f as follows. For all $0 \le i \le n$, add the rules given by Fig. 4.[3] Finally we require that the top level formula f should hold in the initial marking

$$\leftarrow \text{not } f(0) \; . \tag{12}$$

With this program $\Pi_{\text{LTL}}(f, n)$ we get our main main result.

Theorem 3. *Let f be an LTL formula in positive normal form and $N = \langle P, T, F \rangle$ be a 1-safe and deadlock free P/T-net for all initial markings satisfying a condition C_0. If $\Pi_{\text{M}}(C_0, 0) \cup \Pi_{\text{A}}(N, n) \cup \Pi_{\text{LTL}}(f, n)$ has a stable model, then there is an execution of N from an initial marking satisfying C_0 which satisfies f.*

[3] An equivalence explaining the release translation: $f_1 \, R \, f_2 \equiv (\Box f_2) \vee (f_2 \, U \, (f_2 \wedge f_1))$.

The size of the program in Theorem 3 is linear in the size of the net and formula, i.e., $\mathcal{O}((|P| + |T| + |F| + |f|) \cdot n)$. The semantics of LTL is defined over interleaving executions. A novelty of the translation is that it allows concurrency between invisible transitions.

Forcing interleaving semantics. We can create the interleaving semantics versions of bounded model checking problems by adding a set of rules $\Pi_{\mathrm{I}}(N, n)$. It includes for each time step $0 \leq i \leq n - 1$ a rule

$$\leftarrow 2\{t_1(i), \ldots, t_m(i)\} \tag{13}$$

where $\{t_1, \ldots, t_m\}$ is the set of all transitions. These rules eliminate all stable models having more than one transition firing in a step.

Corollary 1. *Let $\Pi_{\mathrm{S}}(N, n)$ be a program solving a bounded model checking problem in the step semantics using any of the translations above. Then the program $\Pi_{\mathrm{S}}(N, n) \cup \Pi_{\mathrm{I}}(N, n)$ solves the same problem in the interleaving semantics.*

3.3 Relation to previous work

In previous work on bounded model checking little attention has been given to the knowledge representation problem of encoding succinctly the unfolded behavior and the temporal property. We address this problem and develop an encoding of the behavior of an asynchronous system which is linear in the size of the system description (Petri net) and in the number of steps. Moreover, it allows the exploitation of the inherent concurrency of the system in model checking.

Our approach could be used as a basis for a similar treatment using propositional logic and satisfiability (SAT) checkers. For simple temporal properties such as reachability and deadlock this is fairly straightforward to develop using the ideas of Clark's completion and Fages' theorem [7]. This is because our encoding produces acyclic programs except for the choice rules which need a special treatment. To achieve a compact SAT encoding is more challenging because propositional logic lacks cardinality constraint rules (2). Their mapping to propositional formulae can result to a quadratic blow-up which is sometimes significant as conflicts may involve even hundreds of transitions.

For general LTL model checking a succinct SAT encoding is challenging. The compactness of our encoding is due to the fact that stable model semantics supports the smallest fixed point evaluation of recursive rules which is exploited in translating the U and R operators. Because of these recursive rules a similar compact SAT encoding is not immediate. In [1] a SAT encoding is given. However, it is more complicated than our linear size encoding but remains polynomial.

4 Experiments

We have implemented the deadlock detection and LTL model checking translations presented in the previous section. The translation is given a fixed initial marking M_0, which allows the following optimizations to be implemented:

Problem	$\|P\|$	$\|T\|$	St. n	St. s	Int. n	Int. s	States
DARTES(1)	331	257	32	0.5	32	0.5	>1500000
DP(6)	36	24	1	0.0	6	0.1	728
DP(8)	48	32	1	0.0	8	0.3	6560
DP(10)	60	40	1	0.0	10	3.3	59048
DP(12)	72	48	1	0.0	12	617.4	531440
ELEV(1)	63	99	4	0.0	9	0.4	163
ELEV(2)	146	299	6	0.5	12	3.9	1092
ELEV(3)	327	783	8	5.6	15	139.0	7276
ELEV(4)	736	1939	10	157.2	>13	1215.2	48217
HART(25)	127	77	1	0.0	>5	1.0	>1000000
HART(50)	252	152	1	0.0	>5	5.7	>1000000
HART(75)	377	227	1	0.0	>5	15.5	>1000000
HART(100)	502	302	1	0.0	>5	35.9	>1000000
KEY(2)	94	92	>25	1937.9	>26	56.1	536
MMGT(3)	122	172	7	11.1	10	87.2	7702
MMGT(4)	158	232	8	687.3	>11	1874.1	66308
Q(1)	163	194	9	0.1	>17	2733.7	123596

Fig. 5. Experiments

- Place and transition atoms are added only from the time step they can first appear on. Only atoms for places $p(0)$ in the initial marking are created for time $i = 0$. Then for each $0 \leq i \leq n - 1$: (i) Add transition atoms for all transitions $t(i)$ such that all the place atoms in the preset of $t(i)$ exist. (ii) Add place atoms for all places $p(i + 1)$ such that either the place atom $p(i)$ exists or some transition atom in the preset of $p(i + 1)$ exists.
- Duplicate rules are removed. Duplicates can appear in (5),(7).

As benchmarks we use a set of deadlock detection benchmarks collected by Corbett [4], converted to 1-safe P/T-nets by Melzer and Römer [14]. The models were picked from those which have a deadlock. For each model and both semantics we incremented the used bound until a deadlock was found. We report the time for `Smodels` to find the first stable model using this bound. In some cases a model could not be found within a reasonable time in which case we report the time used to prove that there is no deadlock within the reported bound. Unfortunately, we did not have a large collection of LTL model checking examples, and benchmarking the LTL translation is left for further work. The experimental results can be found in Fig. 5. The columns are:

- Problem: The problem name with the size of the instance in parenthesis.
- $|P|$: Number of places in the original net.
- $|T|$: Number of transitions in the original net.
- St. n: The smallest integer n such that a deadlock could be found using the step semantics / in case of $> n$ the largest integer n for which we could prove that there is no deadlock within that bound using the step semantics.
- St. s: The time in seconds to find the first stable model / to prove that there is no stable model. (See St. n above.)
- Int. n and Int. s: defined as St. n and St. s but for the interleaving semantics.
- States: Number of reachable states of the P/T-net (if known).[4]

[4] These differ from the ones reported in [11] where unfortunately there are some errors.

The times reported are the average of 5 runs of the time for `smodels 2.26` as reported by the `/usr/bin/time` command on a 450Mhz Pentium III PC running Linux. The used tools, nets, and logic programs are available from: <`http://www.tcs.hut.fi/~kepa/experiments/LPNMR2001/`>.

In many of the experiments the step semantics version found a deadlock with a smaller bound than the interleaving one. Also, when the bound needed to find the deadlock was fairly small, the bounded model checker was performing well. In the examples ELEV(4), HART(x) and Q(1) we were able to find the counterexample only when using step semantics. In the KEY(2) example we were not able to find a counterexample with either semantics, even though the problem is known to have only a small number of reachable states. In contrast, the DARTES(1) problem has a large state-space, and despite of it a counterexample of length 32 was obtained. Overall, the results are promising, in particular, for small bounds and the step semantics.

5 Conclusions

We introduce bounded model checking of asynchronous concurrent systems modeled by 1-safe P/T-nets as an interesting application area for answer set programming. We present mappings from bounded reachability, deadlock detection and LTL model checking problems of 1-safe P/T-nets to stable model computation. Our approach is capable of doing model checking for a set of initial markings at once. This is usually difficult to achieve in current enumerative model checkers and often leads to state space explosion. We handle asynchronous systems using a step semantics whereas previous work on bounded model checking only uses the interleaving semantics [1]. Furthermore, our encoding is more compact than the previous approach employing propositional satisfiability [1]. This is because our rule based approach allows to represent executions of the system, e.g. frame axioms, succinctly and supports directly the recursive fixed point computation needed to evaluate LTL formulae.

The first experimental results indicate that stable model computation is quite a competitive approach to searching for short executions of the system leading to deadlock and worth further study. More experimental work and comparisons are needed to determine the strength of the approach. In particular, for comparing with SAT checking techniques, it would be interesting to develop a similar treatment of asynchronous systems using a SAT encoding and compare it to the logic program based approach.

Relating the net unfolding method (see [9, 14] and further references there) to bounded model checking would be interesting. There are also alternative semantics to the two presented in this work [10], applying them to bounded LTL model checking is left for further work.

References

1. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'99), pages 193–207. Springer, March 1999.

2. J. Burch, E. Clarke, K. McMillan, D. Dill, and L.Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.
4. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. Technical report, Department of Information and Computer Science, University of Hawaii at Manoa, 1995.
5. J. Desel and W. Reisig. Place/Transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, pages 122–173. Springer-Verlag, 1998.
6. 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.
7. F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
8. 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.
9. 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.
10. K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (Concur'2001)*, Aalborg, Denmark, August 2001. Accepted for publication.
11. K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 90–96, Stanford, USA, March 2001. AAAI Press, Technical Report SS-01-01.
12. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
13. O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proceeding of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 172–183. Springer-Verlag, 1999.
14. 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.
15. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
16. I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.
17. K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 472–475, Haifa, Israel, June 1997. Springer-Verlag.