TU Clausthal
Clausthal University of Technology

Model Checking with Logic Based Petri Nets

Tristan M. Behrens and Jürgen Dix

IfI Technical Report Series **IfI-07-02**

IfI

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Model Checking with Logic Based Petri Nets

Tristan M. Behrens and Jürgen Dix

Department of Informatics, Clausthal University of Technology
Julius-Albert-Straße 4, 38678 Clausthal, Germany
{behrens,dix}@in.tu-clausthal.de

Abstract

We introduce a new class of Petri nets, simple logic Petri nets (*SLPN*). We show how this type of nets can be efficiently mapped into logic programs with negation: the corresponding answer sets describe interleaved executions of the underlying Petri nets. We also show how to model and specify AgentSpeak agents with *SLPN*'s. This allows us to solve the task of *model checking AgentSpeak agents* by computing answer sets of the obtained logic program with any ASP system.

1 Introduction

The problem of design validation—ensuring the correctness of a design as early as possible—has long been a major challenge for software engineers. This is also true for developers of *multi agent systems*. Classical techniques like *simulation* and *testing* do not to scale up (measured in the size of the underlying systems). Modern techniques of *formal verification* have the problem of scaling up as well, but some of them seem to be very promising: in particular *theorem proving* and *model checking*.

Model checking is the exhaustive enumeration of all possible executions of a system (or model) in order to prove desired behavioral properties. While techniques relying on theorem provers are mostly semi-automatic (and require a lot of human interference) model checking is often fully automatic and leads to counterexamples in case correctness can not be shown. These models usually give the designer of the system important information and are used to improve the design of it.

Petri nets—bipartite graphs with a state-transforming function—are such formal models for the description and the execution of *concurrent systems*.

The main ideas introduced in this paper are (1) to introduce a new kind of Petri net, (2) to use these nets for *specifying* agents, and (3) to transform these nets into logic programs and use ASP engines ([1, 4, 3, 5]) for solving the model-checking task.

Our nets are based on *logical literals* and their creation and absorption in the respective topology. We claim that such nets are perfectly suited for systems where atoms and literals are the main data-structures (like languages based on variants of AgentSpeak).

- We introduce the class *SLPN* of *Simple Logic* Petri nets.

- We show their relation to *Answer Set Programming*, by providing a transformation from *SLPN* into logic programs with negation.
- While we use smodels as our inference engine, our approach does not depend on it and we can use other engines as well (experiments are on their way).
- This transformation is particularly useful when it comes to bounded model checking of our nets.
- We show their applicability to *Agent Oriented Programming* and show how we can model agents written in a well known agent-oriented programming language (AgentSpeak). We believe our nets can be extended to cover much more agent-oriented programming languages.
- Finally we discuss related work and give an outlook to future work.

2 Basic Terminology

DEFINITION 1 (Terms)**.** *Let VAR be a finite set of* variables*, $CONST$ be a finite set of* constants *and $FUNC$ be a finite set of* function-symbols.

The set of terms *is the smallest set $TERM$ for which the following holds: (1) For each $var \in VAR$, $var \in TERM$. (2) For each $const \in CONST$, $const \in TERM$. (3) For each $f \in FUNC$ of arity i and $u_1, \ldots, u_i \in TERM$, $f(u_1, \ldots, u_i) \in TERM$.*

Note that the set $TERM$ is infinite, even though the sets VAR and $CONST$ are finite. We are often using the constant 1 and the binary function symbol $+$ to have available terms $1, 1+1, 1+1+1, \ldots$ which we identify with the natural numbers.

As is custom in Prolog-like languages, for variables we use strings starting with upper-case letters and for constants we allow either numbers or identifiers which are strings starting with a lower-case letter. We use the common terminology for functions, but we use the infix-notation for the standard arithmetical functions for the sake of readability.

DEFINITION 2 (Literals, Groundedness)**.** *Let $PRED$ be a set of predicate symbols. The following sets are called sets of* positive *(resp.* negative*)* propositional literals*.*

$$A^+ = \{p(t_1, \ldots, t_n) \mid p \in PRED, t_i \in TERM, n \in \mathbb{N}_0\}$$
$$A^- = \{\neg p(t_1, \ldots, t_n) \mid p \in PRED, t_i \in TERM, n \in \mathbb{N}_0\}$$

- *$L = A^+ \cup A^-$ is the set of* propositional literals.
- *For all $X \subseteq L$ we denote with X_{grnd} all* ground *elements of X, i.e. those that do not contain variables.*
- *$var\text{-}of : 2^L \to 2^{VAR}$ selects all variables that are contained in a given set of literals.*

When a literal contains no terms we omit the brackets. We also use $a(\vec{t})$ instead of $a(t_1, \ldots, t_n)$ for ease of reading.

3 Simple Logic Petri Nets

Petri nets are often referred to as *token games* and are usually defined by first providing the topology and then the semantics. The topology is a net structure, i.e. a bipartite digraph consisting of *places* and *transitions*. Defining the semantics means (1) defining what a state is, (2) defining when a transition is enabled and, finally, (3) to define how tokens are consumed/created/moved in the net. We refer to the notion that tokens are consumed and created by *firing transitions*—moving is made possible by *consuming* and *creating*. Our most basic definition is as follows:

DEFINITION 3 (Simple Logic Petri Net)**.** *The tuple* $\mathcal{N} = \langle P, T, F, C\rangle$ *with*

$P = \{p_1, \dots, p_m\}$ *the set of places,*

$T = \{t_1, \dots, t_n\}$ *the set of transitions,*

$F \subseteq (P \times T) \cup (T \times P)$ *the inhibition relation,*

$C : F \rightarrow 2^L$ *the capacity function,*

is called a Simple Logic Petri Net *(SLPN).*

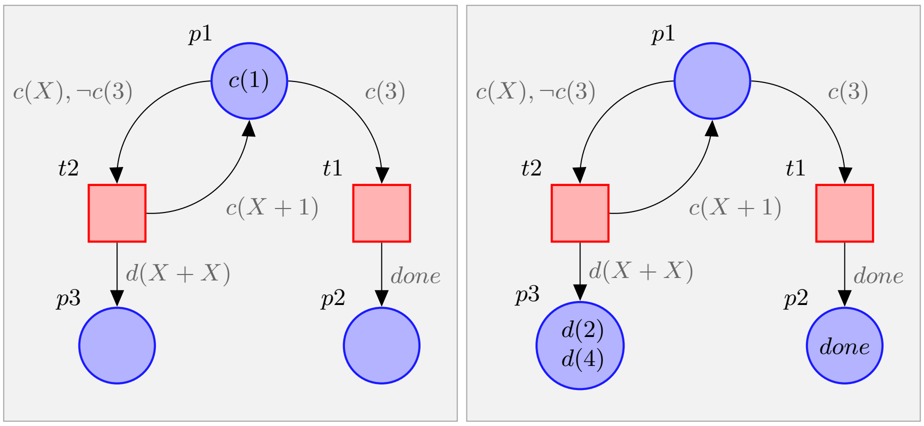

Figure 1: A Simple Logic Petri Net (*SLPN*) example. On top the initial state. Below the state after all enabled transitions were fired.

EXAMPLE 1 (Running example (1))**.**
Figure 1 shows a SLPN in two states of execution. Places are depicted as circles, transitions are as boxes, arcs as arrows. The only value is 1*, the only binary operator is* +*. The set of ground terms is* $\{1, 1+1, 1+1+1, \dots\}$ *which we identify with* $\mathbb{N}$*. The net structure is* $P = \{p_1, p_2, p_3\}$*,* $T = \{t_1, t_2\}$*, and*

$$F = \{(p_1, t_1), (p_1, t_2), (t_1, p_2), (t_2, p_1), (t_2, p_3)\}$$

The capacity is defined as

$$\begin{aligned} C(p_1, t_1) &= \{c(3)\} \\ C(p_1, t_2) &= \{c(X), \neg c(3)\} \\ C(t_2, p_1) &= \{c(X+1)\} \\ C(t_1, p_2) &= \{done\} \\ C(t_2, p_3) &= \{d(X+X)\} \end{aligned}$$

The next definition is helpful when formulating claims about *SLPN*'s.

DEFINITION 4 (Preset, Postset)**.** *For each $p \in P$ and each $t \in T$ we define the preset and postset of p and t as follows*

$$\begin{aligned} \bullet p &= \{t \in T \mid (t,p) \in F\} \\ p \bullet &= \{t \in T \mid (p,t) \in F\} \\ \bullet t &= \{p \in P \mid (p,t) \in F\} \\ t \bullet &= \{p \in P \mid (t,p) \in F\} \end{aligned}$$

EXAMPLE 2 (Running example (2))**.**
We have the following presets and postsets:

$$\begin{array}{ll} \bullet p_1 = \{t_2\} & p_1\bullet = \{t_1, t_2\} \\ \bullet p_2 = \{t_1\} & p_2\bullet = \emptyset \\ \bullet p_3 = \{t_2\} & p_3\bullet = \emptyset \\ \bullet t_1 = \{p_1\} & t_1\bullet = \{p_2\} \\ \bullet t_2 = \{p_1\} & t_2\bullet = \{p_1, p_3\} \end{array}$$

Before we take a look at the semantics we impose two restrictions on the creation of *SLPN*'s: Firstly, if a variable occurs in the label of an **outgoing** arc from a transition, then this variable must also occur in an **ingoing** arc to this transition. While this condition resembles the well-known *safeness*-property in databases, it is weaker, because variables can also occur negatively in incoming arrows. Secondly, we do not allow negative atoms as labels of arcs between transitions and places. Otherwise parts of the net would not be executable or would make no sense at all: This is a variant of CWA.

DEFINITION 5 (Valid Net)**.** *A SLPN $\mathcal{N} = \langle P, T, F, C \rangle$ is* valid *iff the following hold:*

$$\forall t \in T: \; var\text{-}of\left(\bigcup_{p \in t\bullet} C(t,p)\right) \subseteq var\text{-}of\left(\bigcup_{p \in \bullet t} C(p,t)\right) \tag{1}$$

$$\forall (t,p) \in F: \; C(p,t) \in A^{+} \tag{2}$$

This means that all variables that are in the labels of arcs between each transition and its postset are also in the labels of the arcs between each transition and its preset. Otherwise we would have uninitialized variables, which corresponds to unsafeness in logic programs.

Now we have places and transitions and the arcs between them, together with a function which labels each arc with a literal. We would like to have grounded atoms inside the places and the ability to move them through the net. Thus we define the state of the net, which is a mapping from P to subsets of A^+_{grnd}:

DEFINITION 6 (State)**.** *A* state *is a function* $s : P \rightarrow 2^{A^+_{grnd}}$*.* s_0 *denotes the* initial state*.*

Before describing the transition between states we need the notion of an *enabled transition*. In each state a Petri net might have none, one or several enabled transitions. This holds for our *SLPN*'s as well as for Petri nets in general:

DEFINITION 7 (Enabling of Transitions, Bindings)**.** *A transition $t \in T$ is* enabled *if there is $B \subseteq VAR \times CONST$:*

$$\forall p \in \bullet t \; \forall a(\vec{t}) \in \; C(p,t) \cap A^+ : \; a(\vec{t})_{[B]} \in s(p) \;\; \textit{and}$$
$$\forall p \in \bullet t \; \forall a(\vec{t}) \in \; C(p,t) \cap A^- : \neg a(\vec{t})_{[B]} \notin s(p)$$

B is a (possibly empty) set of variable substitutions. We denote wlog by $Subs(t) = \{B_1, \ldots, B_n\}$ with $n \in \mathbb{N}$ the set of all sets of such variable substitutions with respect to the transition t for which the above holds.

This means that a transition t is enabled if (1) all positive literals that are labels of arcs between t and its preset $\bullet t$ are unifiable with the literals in the respective places, and (2) that there is no unification for all the negative literals that are labels of arcs between the transition and its preset. Note that we need $\neg a$ in the second formula, because the places never contain negative atoms (closed world assumption).

We are now ready to define transitions of states. *Firing transitions* absorb certain atoms from the places in the preset and put new atoms into the places in the postset using the variable substitutions:

DEFINITION 8 (State Transition)**.**

$$\forall p \in P : \; s'(p) = s(p) \;\setminus\; \Big(\bigcup_{t \in p\bullet, t fires} C(p,t)_{[Subs(t)]}\Big) \;\cup\; \Big(\bigcup_{t \in \bullet p, t fires} C(t,p)_{[Subs(t)]}\Big)$$

Thus each place receives ground literals from each firing transition in its preset and ground literals are absorbed by all the firing transitions in the postset, thus leading to a new state of the whole system.

EXAMPLE 3 (Running example (3))**.**
Figure 1 shows a Petri net in two states. The initial state is $s_0(p_1) = \{c(1)\}$, $s_0(p_2) = s_0(p_3) = \emptyset$. Let us assume that in each state all enabled transitions fire. This will in the end lead to a final state: $s_f(p_1) = \emptyset$, $s_f(p_2) = \{done\}$, $s_f(p_3) = \{d(2), d(4)\}$.

4 ASP Representation

In this section we concentrate on the *answer set representation* of the introduced Petri net type. We have chosen ASP because it is very promising logic programming formalism that combines a declarative language with several existing and very efficient inference engines. Moreover, *SLPN*'s and logic programs share certain similarities. The creation and consumption of atoms in places over time can easily be depicted by (1) adding an annotation for the place in which it is located and (2) adding another annotation to mark the respective timestep.

We show that any *SLPN* $\mathcal{N}$ can be transformed into a logic program with negation $\mathtt{Trans}(\mathcal{N})$ such that the answer sets of the latter correspond to the different interleaved executions of the concurrent system represented by $\mathcal{N}$.

Our approach can be used with any ASP engine: the transformation from a simple logic Petri Net $\mathcal{N}$ into a logic program $\mathtt{Trans}(\mathcal{N})$ can be fed into any such engine and thus can profit from the growing availability of ASP systems and their ongoing improvements.

We are currently undertaking experiments with one particular ASP engine, *smodels*, for two reasons. (1) *smodels* seems to be promising when it comes to speed-considerations ([9]). In addition, it has been shown that model checking a basic type of Petri nets can be reduced to planning ([8]). (2) *smodels* supports several aggregation functions that might be used to optimize our transformation.

Before describing the mapping $\mathtt{Trans}$ in detail, we introduce some terminology:

DEFINITION 9 (Positive Labels, Negative Labels)**.** *Let $\mathcal{N} = \langle P, T, F, C\rangle$ be a simple logic Petri net. We call the set $A_{t,N} = \left\{a(\vec{t}, p, N) \mid a(\vec{t}) \in C(p,t)\right\}$ the set of* positive labels *wrt. timestep N. We call the set $B_{t,N} = \left\{b(\vec{t}, p, N) \mid \neg b(\vec{t}) \in C(p,t)\right\}$ the set of* negative labels *wrt. timestep N. Wlog we denote these sets by*

$$A_{t,N} = \left\{a_1(\vec{t_{a_1}}, p_{a_1}, N), \dots a_m(\vec{t_{a_m}}, p_{a_m}, N)\right\},\ m \in \mathbb{N}_0,$$
$$B_{t,N} = \left\{b_1(\vec{t_{b_1}}, p_{b_1}, N), \dots a_n(\vec{t_{n_m}}, p_{b_n}, N)\right\},\ m \in \mathbb{N}_0.$$

The positive and negative labels are important: they correspond to atoms with appropriate annotations for the places and steps. More precisely, we use rules according to the following guidelines:

DEFINITION 10 (Transforming a *SLPN* into a logic program)**.**
Given a SLPN $\mathcal{N} = \langle P, T, F, C\rangle$ we define a logic program $\mathtt{Trans}(\mathcal{N}) = \mathtt{Trans}_0(\mathcal{N}) \cup \mathtt{Trans}_1(\mathcal{N}) \cup \mathtt{Trans}_2(\mathcal{N}) \cup \mathtt{Trans}_3(\mathcal{N})$. In the following, we describe how to construct the sets of rules $\mathtt{Trans}_i(\mathcal{N})$. We use the predicates $enabled(t, N)$, $fires(t, N)$, and $erase\_a(t, p, N)$ (for all terms $a(\dots)$ occurring as labels). We also use 0, 1 and the binary function term $+$ to denote timepoints $0, 1, 2, \dots$.

EXAMPLE 4 (Running example (4))**.**
How do we construct a logic program from the SLPN in Fig. 1? We get for the initial

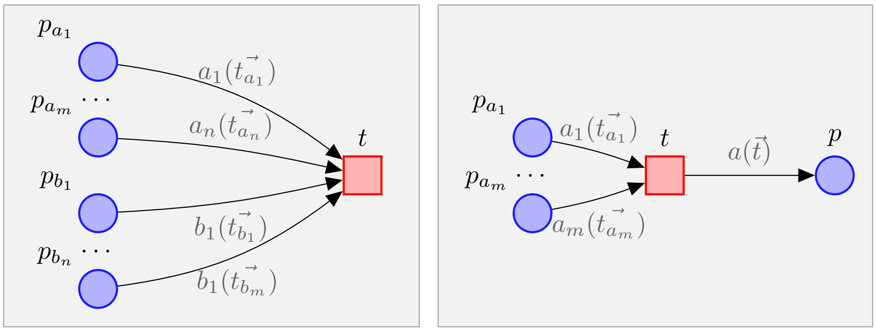

Figure 2: Enabling transitions (left) and spawning tokens (right). Schematic views.

state: $\mathtt{Trans}_0(\mathcal{N}) := \{c(1, p1, 0)\}$. *We enable transitions by defining* $\mathtt{Trans}_1(\mathcal{N})$ *to consist of the following two rules:*

$$\begin{array}{lcl} enabled(t1, N) & \leftarrow & c(3, p1, N), \\ enabled(t2, N) & \leftarrow & c(X, p1, N), \neg c(3, p1, N) \end{array}$$

For the nondeterministic firing of the enabled transitions we have $\mathtt{Trans}_2(\mathcal{N})$*:*

$\{fires(t1, N)\} \leftarrow enabled(t1, N)$.
$\{fires(t2, N)\} \leftarrow enabled(t2, N)$.
$\leftarrow \neg fires(t1, N), \neg fires(t2, N)$.

The last rule ensures that in each step at least one transition fires. The notation $\{fires(t1, N)\} \leftarrow enabled(t1, N)$ *is a* choice rule*: it means that* $fires(t1, N)$ *can be true or not, if* $enabled(t1, N)$ *is true.*

Last but not least we have to take the tokens into account. We have 6 labels (done, $c(3)$, $\neg c(3)$, $d(X + X)$, $c(X)$, $c(X + 1)$*) and therefore corresponding predicates erase_d, erase_c, erase_done.* $\mathtt{Trans}_3(\mathcal{N})$ *consists of three group of rules:*

Created tokens:

$done(p2, N + 1) \leftarrow fires(t1, N), c(3, p1, N)$.
$c(X + X, p3, N + 1) \leftarrow fires(t2, N), c(X, p1, N)$.
$c(X + 1, p1, N + 1) \leftarrow fires(t2, N), c(X, p1, N)$.

Consumed tokens:

$erase\_c(3, p1, N) \leftarrow c(3, p1, N), fires(t1, N)$.
$erase\_c(X, p1, N) \leftarrow c(X, p1, N), fires(t2, N)$.

Kept tokens:

$done(P, N + 1) \leftarrow done(P, N), \neg erase\_done(P, N)$.
$d(X, P, N + 1) \leftarrow d(X, P, N), \neg erase\_d(X, P, N)$.
$c(X, P, N + 1) \leftarrow c(X, P, N), \neg erase\_c(X, P, N)$.

In the above rules, N is used as a variable ranging over $\{1, \ldots, n\}$ for a fixed $n \in \mathbb{N}$. For $n = 1, \ldots, 4$ we get the following answer sets s_i:

$$
\begin{aligned}
s_1 &= \{c(1, p1, 1), c(2, p1, 2), d(2, p3, 2)\} \\
s_2 &= \{c(1, p1, 1), c(2, p1, 2), c(3, p1, 3), d(2, p3, 2), \\
&\quad d(2, p3, 3), d(4, p3, 3)\} \\
s_3 &= \{c(1, p1, 1), c(2, p1, 2), c(3, p1, 3), c(4, p1, 4), \\
&\quad d(2, p3, 2), d(2, p3, 3), d(4, p3, 3), d(2, p3, 4), \\
&\quad d(4, p3, 4), d(6, p3, 4)\} \\
s_4 &= \{c(1, p1, 1), c(2, p1, 2), c(3, p1, 3), c(4, p1, 4), \\
&\quad d(2, p3, 2), d(2, p3, 3), d(4, p3, 3), d(2, p3, 4), \\
&\quad d(4, p3, 4), d(6, p3, 4), d(2, p3, 5), d(4, p3, 5), \\
&\quad d(6, p3, 5), done(p2, 5)\}
\end{aligned}
$$

Finally, for s_5 no answer set exists. This corresponds to a deadlock. When no transitions are activated (in step 5), then the constraint $\leftarrow \neg fires(t1, 5), \neg fires(t2, 5)$ can not be satisfied. The s_i are answer sets that correspond to the interleaved executions of the SLPN. s_4 shows an execution that reaches the final state shown in Fig. 1. s_5 is empty because no transitions can fire.

In the above rules, the variable N plays a special role. In fact, we let N ranges over $\{1, \ldots, n\}$) for *fixed* n and then increment n until a deadlock occurs (no answer sets exist anymore). For fixed n, we introduce the notation $\mathtt{Trans}(\langle \mathcal{N}, n \rangle)$ to make explicit the dependency of the transformation on the range of N. Our main theorem states that the transformation $\mathtt{Trans}$ is sound and complete:

THEOREM 1 (*SLPN* vs. ASP)**.**
Given a simple logic Petri net $\mathcal{N}$ and its transformation into a logic program $\mathtt{Trans}(\mathcal{N})$ as defined below, the following holds:

1. *Each answer set of $\mathtt{Trans}(\langle \mathcal{N}, n \rangle)$ is equivalent to one interleaved execution of the concurrent system represented by the SLPN's.*

2. *Each interleaved execution of the concurrent system represented by the SLPN's is represented by an answer set of $\mathtt{Trans}(\langle \mathcal{N}, n \rangle)$.*

Thus the answer sets of $\mathtt{Trans}(\langle \mathcal{N}, n \rangle)$ represent all possible executions after at most n activations: they do not lead to a deadlock. The deadlock occurs when n is large enough so that no answer set exists (in our running example this is s_5).

The precise definition of $\mathtt{Trans}(\mathcal{N})$ is as follows.

- $\mathtt{Trans}_0(\mathcal{N})$, **initial state:** $\forall p \in P\ \forall a(\vec{t}) \in s_0(p)$ introduce predicates $a(\vec{t}, p, 0)$.

- $\mathtt{Trans}_1(\mathcal{N})$, **enabling transitions:** $\forall t \in T$ introduce the following rules

  $$\begin{aligned} enabled(t, N) \leftarrow\ & a_1(\vec{t}_{a_1}, p_{a_1}, N), \ldots, a_m(\vec{t}_{a_m}, p_{a_m}, N), \\ & \neg b_1(\vec{t}_{b_1}, p_{b_1}, N), \ldots, \neg b_n(\vec{t}_{b_n}, p_{b_n}, N). \end{aligned}$$

- $\mathtt{Trans}_2(\mathcal{N})$, **firing transitions:** For $\{t_1, \ldots, t_2\} = T$ introduce the constraint

  $$\leftarrow \neg fires(t_1, N), \ldots, \neg fires(t_n, N).$$

  and the choice rules $\{fires(t_i, N)\} \leftarrow enabled(t_i, N)$.

- $\mathtt{Trans}_3(\mathcal{N})$, **tokens:** this consists of 3 groups of rules

  - **creating tokens:** $\forall p \in P\ \forall t \in \bullet p\ \forall a(\vec{t}) = C(t, p)$ with $A_{t,N}$ as defined above introduce the rules

    $$a(\vec{t}, p, N + 1) \leftarrow fires(t, N),\ a_1(\vec{t}_{a_1}, p_{a_1}, N), \ldots, a_m(\vec{t}_{a_m}, p_{a_m}, N).$$

  - **consuming tokens:** $\forall a(\vec{t}) \in \bigcup_{p \in P, t \in T} C(p, t)$ introduce the rules

    $$erase\_a(\vec{t}, p, N) \leftarrow a(\vec{t}, p, N),\ fires(t, N).$$

  - **kept tokens:** $\forall a \in PRED$ with appropriate arity introduce the rules

    $$a(T_1, \ldots, T_n, P, N + 1) \leftarrow a(T_1, \ldots, T_n, P, N), \neg erase\_a(T_1, \ldots, T_n, P, N).$$

For ease of reading, we do not depict the necessary restrictions of variables in some of the rules: N is usually a time step, P and T represent places and transitions.

Proof. Let AS be an answer set of the logic program $\mathtt{Trans}(\langle \mathcal{N}, n \rangle)$.

$\mathtt{Trans}(\mathcal{N}_i)$ for $i = 0, \ldots 2$ clearly describe the initial state and the enabled/firing transitions (the choice rules describe all possibilities).

The first rule in $\mathtt{Trans}_3(\mathcal{N})$ adds atoms to the postset of a firing transition. The variable bindings are the same as in $\mathtt{Trans}_1(\mathcal{N})$. The second rule in $\mathtt{Trans}_3(\mathcal{N})$ marks atoms for removal if a transition in the postset fires. The third rule in $\mathtt{Trans}_3(\mathcal{N})$ keeps atoms in their places if they are not marked for removal.

Note that there is no real recursion through negation: although the predicate $erase$ depends negatively on itself (through the last two rules in $\mathtt{Trans}_3(\mathcal{N})$) the third argument, N, is incremented. The only possibility that no answer set exists, is when the constraint in $\mathtt{Trans}(\mathcal{N}_2)$ can not be satisfied, because no transitions are activated. This certainly happens for large enough n. Therefore the $fires$ predicate is not true for such n. Thus the last rule in $\mathtt{Trans}_2(\mathcal{N})$ ensures that there are no answer sets.

Now let any interleaved execution of the concurrent system represented by the *SLPN*'s be given. We assume that for this execution there are exactly n activations (and no deadlock yet). For such an execution, there must be appropriate activations and firing transitions. But all possible activations and firings are completely described in the logic program $\mathtt{Trans}(\langle\mathcal{N}, n\rangle)$ (the choice rules exhaust all possibilities). Thus the execution must correspond to one of the answer sets of $\mathtt{Trans}(\langle\mathcal{N}, n\rangle)$. $\square$

5 Applications

Our aim is to model a variety of established data-structures and algorithmic patterns that are used in computer science. In particular we allow that data can be both accessed locally and remotely in the system. Subnets can function as *queues* or *stacks* whereas at the same time other subnets operate as finite state machines as markers for the state of execution.

It is obvious that we can use our Petri nets to model systems whose *main data-structure consists of logical atoms*. But we can go further than that and move to the realm of multi-agent systems. The idea is to (1) model several agents and an appropriate representation of the environment and then (2) model the channels of correspondence between those—message channels and act/perceive-channels.

For the rest of this paper, we consider an agent, rather general, as any software-entity consisting of (1) a mental state (e.g. beliefs, desires, intentions) and (2) a state of execution.

For multi-agent systems based on logical atoms we could, for example, model agents written in AgentSpeak (F) (a finite subset of AgentSpeak (L)). AgentSpeak agents implement the notion of BDI and can be seen as *reactive planners*: fixed plans are triggered by some event *inside* or *outside* the agent itself. AgentSpeak agents have mental state (beliefs and intentions) and a state of execution (deliberation cycle).

We would like to apply the bounded model checking technique to AgentSpeak (F). Therefore we do the following (see Fig. 3):

1. model an AgentSpeak (F)-MAS as a *SLPN*;
2. apply the bounded model checking technique described in the previous section, thus transforming the *SLPN* into a program to be used in ASP.

5.1 AgentSpeak(F)

We will now concentrate on AgentSpeak. An AgentSpeak (F) agent definition consists of an initial belief base and a set of plans. Each agent in execution uses the following data structures:

- *belief base:* stores the agent's beliefs as a set of atoms,

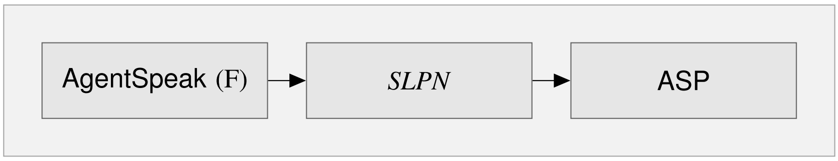

Figure 3: Modelchecking AgentSpeak (F) with ASP. We transform AgentSpeak (F) into *SLPN* and then into ASP.

- *message queue:* stores messages from other agents
- *event queue:* stores so called *triggering events* that might lead to a plan execution,
- *intention list:* stores partially instantiated plans,
- *percepts:* stores perceptions.

In the deliberation cycle of an AgentSpeak (F) agent it repeatedly executes the following segments:

1. **Check Messages**: the agent handles the messages in the inbox and updates the belief base accordingly.
2. **Belief Revision**: the agent updates his beliefs in respect to the percepts. Percepts that are not already in the belief base are added, beliefs that are not in the percepts are removed. In both cases respective events are raised.
3. **Plan Selection**: the agent uses the first element of the event queue and the belief base to detect applicable plans. Applicable plans then generate new intentions which are stored in a free place in the intention list.
4. **Intention Selection**: a *round robin scheduler* selects the next intention in the *Intention Selection segment* and executes the next formula of the plan.

5.2 Modeling AgentSpeak(F)

We now model AgentSpeak (F) agents. For each agent we generate

1. the net $\mathcal{N}_{Cycle}$ representing the deliberation cycle,
2. the subnet $\mathcal{N}_{Message}$ for message handling,
3. the subnet $\mathcal{N}_{BRF}$ for belief revision,
4. the subnet $\mathcal{N}_{Plan}$ for plan selection and
5. the net $\mathcal{N}_{Intention}$ for intention selection and the execution of plan formulæ.

Due to space restrictions we will concentrate only on the (sub)nets $\mathcal{N}_{Cycle}$,$\mathcal{N}_{Plan}$ and $\mathcal{N}_{Intention}$. We only consider the most important part: the execution of plan formulæ. As far as $\mathcal{N}_{Message}$ and $\mathcal{N}_{BRF}$ are concerned, we only sketch their functionality.

5.2.1 Deliberation Cycle

Since the agent infinitely repeats the four phases of the deliberation cycle we model these phases as a finite state machine as depicted in Fig. 4, where the places p_1, p_2, p_3, p_4 represent the phases. The atom tok marks the state of execution (it corresponds to the notion of token we know from the traditional P/T-nets) and triggers the subnets $\mathcal{N}_{Message}$, $\mathcal{N}_{BRF}$, $\mathcal{N}_{Plan}$ and $\mathcal{N}_{Intention}$.

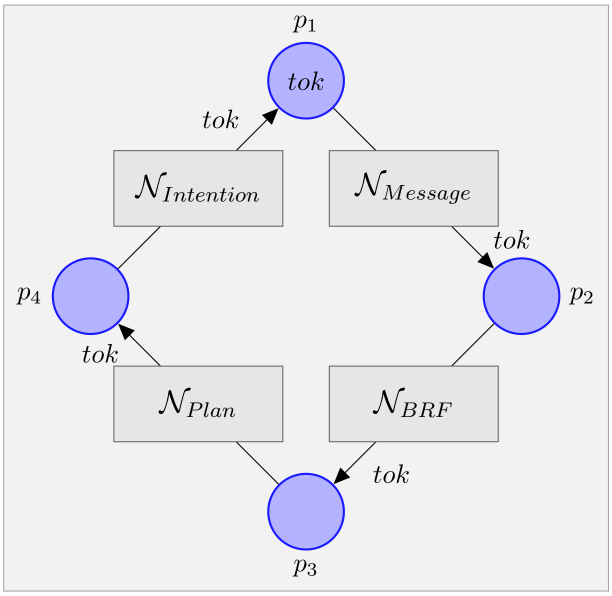

Figure 4: The net $\mathcal{N}_{Cycle}$ represents the AgentSpeak (F) deliberation cycle. The subnets $\mathcal{N}_{Message}$, $\mathcal{N}_{BRF}$, $\mathcal{N}_{Plan}$ and $\mathcal{N}_{Intention}$ stand for the four phases of the deliberation cycle. The token tok is a special one to represent the current state of execution.

5.2.2 Plan Selection

The belief base is modeled as a single place which can be queried and updated on demand. The percepts and the events are modeled as a series of places – each representing one element to establish an order – only the first element can be queried and only the place can be updated which is the first one that is empty.

In order to create a new intention both the belief-base and the first element of the belief base are queried: a plan in AgentSpeak (F) is a construct $e : c \leftarrow h$ where e is a triggering event, c is a conjunction over (negated) beliefs called *context* and h is a sequence of plan fomulæ called *plan head*. A plan is scheduled for execution in the intention list if e is the first element in the event-queue and c is evaluated as being true. Figure 5 shows a *SLPN* that selects the plan $!dress : rain \wedge temp(T) \leftarrow h$. This

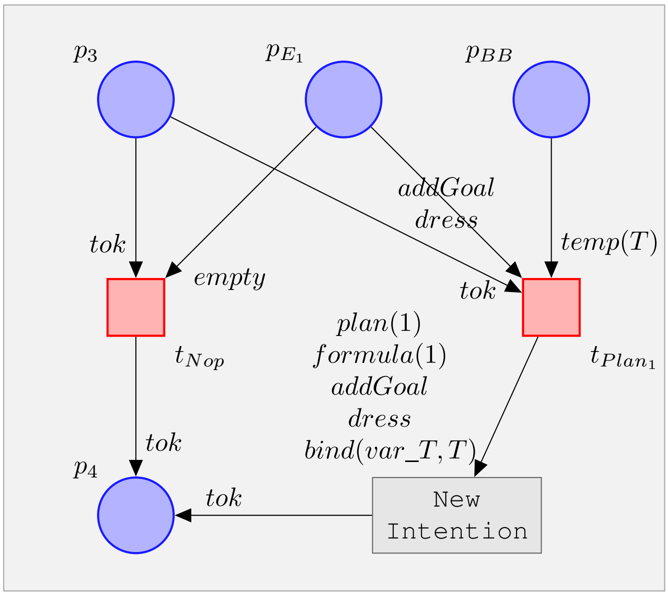

Figure 5: The subnet $\mathcal{N}_{Plan}$ is the plan selector. The plan "$!dress : rain \wedge temp(T) \leftarrow h$" is applicable when the first event in the event queue is the addition of the goal $dress$ and the agent holds the belief that $rain$ and $temp(T)$ are true. The subnet `New Intention` creates a new intention using the triggering event and the binding of the variable T.

means that if the agent adopts the goal *dress*, it considers both the outside temperature and whether it is raining or not. For each plan we have a respective transition (like t_{Plan_1} in the picture)—such a transition is enabled if the plan is applicable. The arc leading from the belief base to the transition represent the plan's context, the arc from the first element in the event-queue represents the plan's triggering event. Note that raising a new intention from an applicable one means the following:

1. store the triggering event;
2. mark the index of the plan;
3. set the plan-formula-counter to 1;
4. copy the variable bindings raised by the triggering event and the context.

5.2.3 Intention Selection

The round robin like scheduler is a finite state machine that uses the atom $index$ (with a term denoting the selected intention which is increased modulo the amount of space in the intention list each time the phase begins).

We assume that we deal with an agent who has three plans with three, one and two plan-formulæ respectively. The plan formula which is to be executed is determined by querying the place p_I for the atoms with the predicates $plan$ and $formula$ as shown in Fig. 6. p_I represents the current intention, selected by the round-robin scheduler mentioned before. This place might contain:

- an atom representing the index to the respective plan (e.g. $plan(2)$),
- an atom representing the index denoting the plan-formula which is to be executed next (e.g. $formula(1)$),
- an atom representing the type of the triggering event ($addBelief$, $remBelief$, $addGoal$ or $remGoal$),
- atoms with the form $bind(varX, val)$ storing variable bindings,
- an optional term-less atom denoting that the intention's execution is suspended ($halt$),
- an optional atom representing the intention that raised the actual intention ($raisedby(1)$).

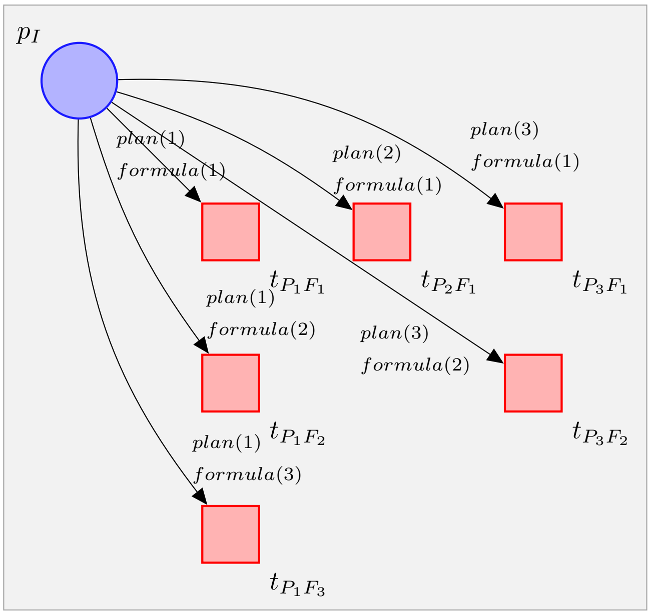

Figure 6: A plan-formula is determined by considering the content of p_I. This picture shows three plans. The first has three formulæ, the second one and the third two.

For the execution of a plan's formula we have to distinguish between:

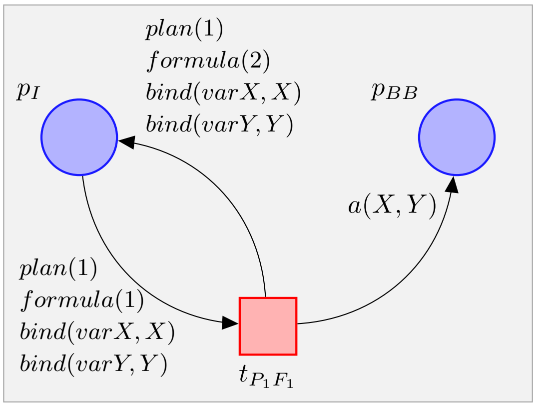

Figure 7: An example for plan formula execution: adding the belief $a(X, Y)$ with the variables replaced to the belief base and increase the formula-counter.

- *add belief*: add the belief to the belief-base and raise a respective event,
- *remove belief*: remove the belief from the belief-base and raise a respective event,
- *test goal*: then update variable bindings according to the belief-base,
- *achievement goal*: raise a new intention and suspend the actual one,
- *action*: then perform the respective action,
- *send message*: put the message into the message-queue of the recipient.

Figure 7 shows a subnet representing the addition of a belief to the agent's belief base. All other subnets would have a similar structure: querying p_I and then updating the respective places and even p_I with new variable bindings and the suspension-atom if necessary.

We still owe the reader sketches of the missing parts of the missing subnets. $\mathcal{N}_{Message}$ queries the message queue for the illocutionary force (type of speech act) and the transmitted data and updates the belief base or adds a goal. And $\mathcal{N}_{BRF}$ queries both the belief base and the percepts queue and adds/removes beliefs as described in our introduction of AgentSpeak (F).

Thus we have sketched how one AgentSpeak (F)-agent can be modeled. Modeling several agents and an environment – the respective nets would be connected (act, perceive, messaging) – would be the fundament for model checking a multiagent system.

6 Related Work

Keijo Heljanko and Ilkka Niemelä presented in their paper [8] a symbolic analysis method for solving bounded deadlock detection and reachability questions for Petri nets using nonmonotonic reasoning techniques. They concentrated on *safe* P/T-nets and obtained very good experimental results. They also did further work on LTL model checking using nonmonotonic reasoning ([9]). Heljanko also implemented *mcsmodels*, which uses finite complete prefixes of safe Petri nets generated by the *PEP tool* [6] in order to do deadlock and reachability checking.

In comparison to our work Niemelä and Heljanko dealt with Petri nets where a state is a mapping from Places into $\{true, false\}$ whereas we map Places into 2^{A^+}. Also, they do not consider the notion of *inhibiting arcs*, which generally *increase* the expressiveness by allowing to prevent transitions from firing. As an example, *if-then-else statements* cannot be expressed within P/T-nets. Additionally, we allow the *comparison* of atoms of the places in the preset of a transition.

CPN-AMI [7] is a collection of tools for modeling, simulating, debugging, structural analysis and model checking of P/T nets and Colored Petri nets. We are interested in using them for structural analysis and model checking tools. We believe that the creation of compatibility between *SLPN* and CPN-AMI is beneficial for formal verification.

In [2], the authors show a mapping from AgentSpeak (F) to Promela. Promela is the input language for the *SPIN Model Checker* [10]. Recently they used *Java Path Finder 2* [12] a verification and testing environment for Java. Using SPIN allowed Bordini et al. to perform unbounded model checking. Because of this our approach is not as powerful as theirs.

7 Future Work

We are currently working on a *SLPN*-framework based on *Petri Net Kernel* (*PNK*) [11]. The developers of *PNK* made a huge effort in making a basic but very useful Petri net infrastructure available for developers and scientists, which allows a smooth development of tools and the exchange of Petri nets in the standardized data exchange format Petri Net Markup Language (PNML).

- We are currently implementing a *SLPN*-to-*smodels* compiler in order to experiment with our transformation and to do some basic model checking with systems defined/modeled in *SLPN*.
- We are also working on an AgentSpeak (F)-to-*SLPN* converter, which allows us to test our transformation.
- We are working on a unbounded *SLPN* model checker for linear time logic that is based on automata theory. With this we could e.g. be able to check temporal properties and the behavior of AgentSpeak (F)-agents.
- We are planning the implementation of a runtime-environment which allows us to execute agents defined in *SLPN* in order to find out if *SLPN* could be a good means to **implement** agents. This will definitely lead to systems which can be model checked easily.

- We are interested in the operational semantics behind *SLPN*'s. Examining this will lead to a deeper understanding of the expressivity of *SLPN*'s and might provide useful means for verification.

- Finally we plan to investigate the relation between *SLPN*'s and other Petri nets. Representing P/T-Nets with Colored Petri nets for example is quite straightforward. We believe that the other direction is possible under certain constraints. This will allow us to use traditional and established Petri net algorithms for our *SLPN*'s.

8 Conclusion

We have introduced with *SLPN* a specialized class of Petri nets with logical atoms as tokens and logical literals as labels. A transition is enabled if the incoming arcs can be unified with the places in the preset. Firing transitions use variable bindings from the preset to spawn net tokens into the preset.

Then we have shown how to transform *SLPN*'s $\mathcal{N}$ into logic programs with negation, such that the interleaved executions of the *SLPN*'s correspond to answer sets of the transformed Petri net $\mathtt{Trans}(\mathcal{N})$. This allows to use existing ASP engines for computing answer sets. Since these ASP engines only allow bounded model checking we will not be able to model check certain properties if the bound was chosen too small.

Finally, we have shown through an example, how to model and specify agents written in AgentSpeak with *SLPN*'s. Thus model checking such agents can be reduced to compute answer sets with current ASP technology. We are currently experimenting with smodels, dlv and other engines to implement the model checking task.

References

[1] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003. ISBN 0521818028.

[2] Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking Agentspeak. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416, New York, NY, USA, 2003. ACM Press.

[3] Gerhard Brewka and Jürgen Dix. Knowledge representation with extended logic programs. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 12, chapter 1, pages 1–85. Reidel Publ., 2. edition, 2005. Shortened version also appeared in Dix, Pereira, Przymusinski (Eds.), *Logic Programming and Knowledge Representation*, Springer LNAI 1471, pages 1–55, 1998.

[4] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In Andrei Voronkov and

Alan Robinson, editors, *Handbook of Automated Reasoning*, pages 1121–1234. Elsevier-Science-Press, 2001.

[5] Jürgen Dix, Ugur Kuter, and Dana Nau. Planning in Answer Set Programming using Ordered Task Decomposition. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay, Volume 1*, pages 521–577. King's College Publications, London, 2005.

[6] Bernd Grahlmann and Eike Best. PEP — more than a Petri net tool. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 397–401. Springer Verlag, 1996.

[7] A. Hamez, Lom Hillah, Fabrice Kordon, Alban Linard, Emmanuel Paviot-Adet, X. Renault, and Yann Thierry-Mieg. New features in CPN-AMI 3: Focusing on the analysis of complex distributed systems. In *ACSD*, pages 273–275. IEEE Computer Society, 2006.

[8] Keijo Heljanko and Ilkka Niemelä. Petri net analysis and nonmonotonic reasoning. In Nisse Husberg, Tomi Janhunen, and Ilkka Niemelä, editors, *Leksa Notes in Computer Science - Festschrift in Honour of Professor Leo Ojala*, pages 7–19, Espoo, Finland, October 2000. Helsinki University of Technology, Laboratory for Theoretical Computer Science.

[9] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *CoRR*, cs.LO/0305040, 2003.

[10] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[11] Ekkart Kindler and Michael Weber. The petri net kernel - an infrastructure for building petri net tools. *International Journal on Software Tools for Technology Transfer*, 3(4):486–497, 2001.

[12] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *ASE*, pages 3–12, 2000.