From PNML to counter systems for accelerating Petri Nets with FAST

Sébastien Bardin[1] and Laure Petrucci[2]

[1] LSV, CNRS UMR 8643
ENS de Cachan
61 avenue du président Wilson
F-94235 CACHAN Cedex
FRANCE
bardin@lsv.ens-cachan.fr
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99 avenue Jean-Baptiste Clément
F-93430 VILLETANEUSE
FRANCE
petrucci@lipn.univ-paris13.fr

Abstract. We use the tool FAST to check parameterized safety properties on Petri nets with a large or infinite state space. Although this tool is not dedicated to Petri nets, it can be used for these as place/transition nets (and some of their extensions) are subcases of FAST input model. The originality of the tool lies in the use of acceleration techniques in order to compute the exact reachability set for infinite systems.
In this paper, we present the automatic transformation of Petri nets written in PNML (Petri Net Markup Language) into counter systems. Then, FAST provides a simple but very powerful language to express complex properties and check these.

1 Introduction

A lot of concurrent models are designed using Petri nets. Many tools are dedicated to Petri nets analysis (e.g. DESIGN/CPN [CPN], the PNK [PNK], ...). However, the analysis often becomes untractable when the state space (also called *reachability set*) of the Petri net is infinite.

In the last few years, a new generation of model-checkers, using *symbolic representations* to handle infinite state spaces, has emerged. The main problem that symbolic model-checking faces is that the reachability set is not computable in the general case. So we can only hope for semi-algorithms. Recently, *acceleration techniques* [BW94,AAB00,FL02] have been developed, which aim at *increasing the convergence* of the reachability set computation. Acceleration computes the effect of iterating a control loop of arbitrary length.

The tool FAST [BFLP03] is dedicated to the analysis of infinite systems with integer variables. FAST uses the automata representation for *semi-linear sets* as a symbolic representation, and the acceleration of control loops described in [FL02]. Moreover, FAST uses a heuristic to find automatically the control loops to be accelerated. Mainly, FAST provides a powerful input model (*counter systems*), the effective computation of the reachability set in most practical cases (about 40 counter systems with infinite reachability sets have been verified and are available at [FAS]) and a *graphical user interface* to guide the user.

We investigate the verification of Petri nets with FAST. Translating common extensions of Petri nets, like nets with inhibitor/read/reset arcs, into counter systems to be analysed by FAST can be done automatically, using an XSLT [Cla99] stylesheet which translates PNML nets [BCvH+03] into counter systems. Then parameterized safety properties can be checked. Several non-trivial Petri nets with infinite state spaces have been analyzed following this method. For more complex nets, such as high-level nets, the translation phase is more tricky, and automatization would require more work on part 2 of the ISO/IEC-15909 International Standard.

This paper is structured as follows. Counter systems verification is presented in section 2. The definition of counter systems, the symbolic representation and the acceleration techniques are introduced. FAST is described in section 3, thus completing the context presentation. Section 4 is the core of this paper. It details the translation of Petri nets described in PNML into counter systems accepted by FAST. Case studies are briefly presented in section 5. Finally, we conclude and future directions of work are given.

2 Verification of Counter Systems

2.1 Counter Systems

Counter systems [FL02] are automata extended with *unbounded integer variables* whose transitions are labelled with *Presburger-linear functions*. A Presburger-linear function is an affine function with a Presburger-definable guard. Recall that the Presburger arithmetics is the first order additive theory $\langle \mathbb{N}, \leq, + \rangle$.

Counter systems are a valuable abstraction since they are a generalization of many well-known models, like Petri nets, Petri nets with inhibitor/read/reset arcs, or Broadcast protocols [EFM99]. In the general case, even reachability properties are undecidable for counter systems.

2.2 Automata Representation for Presburger Sets

Firstly, we need a *good symbolic representation* to represent finitely the infinite reachability sets of counter systems. To check safety properties, this symbolic representation has to be closed under union, intersection and image by a Presburger-linear function, and emptiness has to be decidable.

Presburger sets provide such a good symbolic representation. Presburger sets are sets over $\mathbb{N}^m$ defined by a Presburger formula (also called semi-linear sets). This class of sets can be *represented symbolically by means of automata* [BC96,WB00,Ler03]. This representation provides all the operations needed to check safety properties on counter systems.

2.3 Computing the Reachability Set of a Counter System

For a general counter system, the classical reachability set computation algorithm, consisting in firing one by one the transitions of the system until all the reachable states have been computed, may not terminate. A solution to help convergence is to use *acceleration*. Acceleration allows to compute in one step the exact effect of iterating an arbitrary number of times a control loop of the counter system. In [BW94,Boi98,FL02], Presburger sets are proved to be closed under the acceleration of a Presburger-linear function f, under some algebraic conditions *often met in practice*.

Then the problem comes down to finding the good loops whose accelerations will lead to the reachability set computation. We use a specific heuristic and the *reduction of the number of cycles* given in [FL02] to find these interesting loops.

3 The FAST Tool

FAST [BFLP03] is a tool dedicated to checking safety properties on counter systems. The main issue addressed by FAST is the computation of the *exact* (infinite) state space. On such a complex problem, termination cannot be granted but FAST uses a semi-algorithm which terminates in most practical cases.

3.1 Inputs and Outputs

FAST inputs are both a model and a strategy for the analysis. Outputs are messages indicating whether the system is safe or not. An example of model and associated strategy will be detailled in section 4.3.

The model describes in a natural way the counter system to analyze. Sets of counter systems which communicate via synchronized transitions and shared variables can be specified. The expressiveness is the same as for standalone counter systems, but the specification is easier.

The strategy is the sequence of computations to perform in order to check the validity of the system. The strategy language is a script language which operates on regions (sets of states), transitions and booleans. All usual operators on sets are available, and primitives to compute the reachability set (forward or backward) are provided. Basically, checking a safety property amounts to declare the initial states, compute the reachability set (1), declare the property to check (*the good states*) (2), test if (1) $\subseteq$ (2) and print the result. Only 5 instructions are needed in our language, hence common analysis can be specified easily. Complex strategies can be defined as well, which may guide the tool more efficiently. This is a more flexible framework than restricting inputs to just a system and a property to verify.

3.2 Architecture

The client-server architecture of FAST is presented in figure 1. The server is written in `C++`, while the client is written in `Java`.

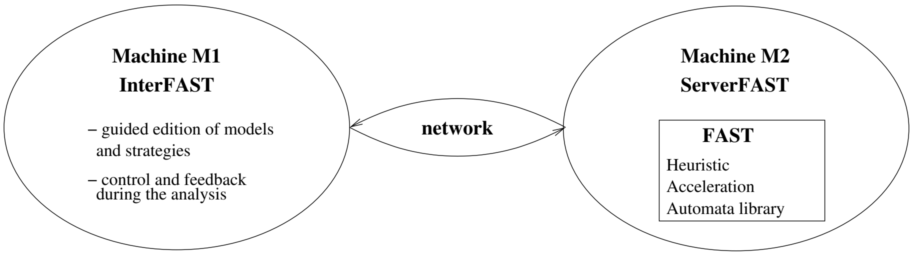

Fig. 1. FAST architecture

- *the server* is the computation engine of FAST. It contains a Presburger library, the acceleration algorithm and the search heuristics.
- client and server communicate through the network via sockets. The computation engine can also be used as standalone.
- *the client* is a front-end which allows the user to interact with the server through a graphical user interface (GUI, figure 2). This interface provides guided edition of models and strategies, with features such as pretty printing or predefined strategies. Once the computation starts, feedback is supplied through different measures and graphs (time elapsed, memory used, . . .).

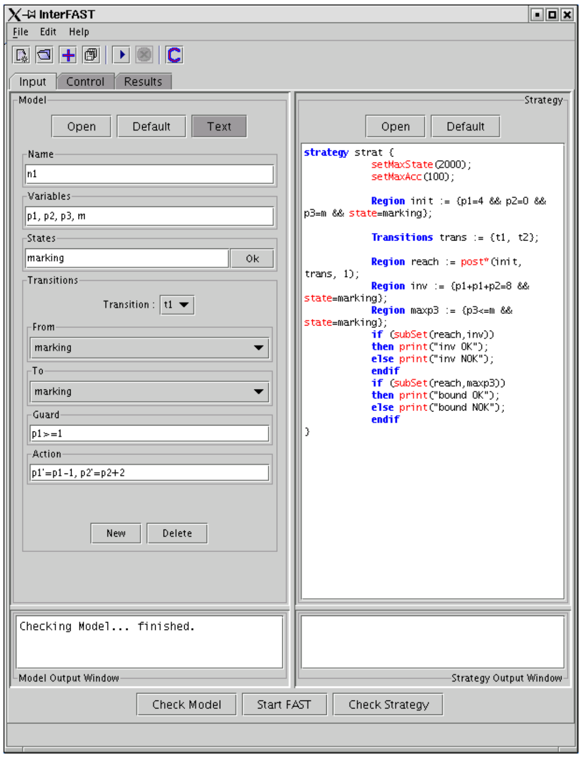

Fig. 2. FAST graphical user interface

3.3 FAST in Short

FAST is a tool both user-friendly and powerful. The GUI, the predefined strategies and the expressiveness of counter systems make FAST easy to use, for both specification and analysis. Strategies give a flexible framework to describe complex analysis, and allow the user to guide the tool when automatic methods fail. Section 4 presents the automatic translation from Petri nets with inhibitor/read/reset arcs from PNML to FAST formats.

4 From Petri Nets to Counter Systems

A lot of concurrent models are designed using Petri nets. Many tools are dedicated to Petri nets analysis (e.g. DESIGN/CPN [CPN], the PNK [PNK], ...), but it often becomes untractable when the state space is infinite.

We aim at using FAST for the analysis of Petri nets with a large or even infinite state space.

4.1 Technique used

The first step towards this goal consists in transforming the Petri net under study into a counter system. For that purpose, we use an approach similar to the one in [BF99]:

- the counter system has only *one state*, which represents a reachable marking of the Petri net ;
- there is *one counter per place* in the net ;
- all *transitions* are loops onto the unique state. Each transition of the system corresponds to a transition in the net. The *guard* associated with a transition checks the enabling condition in the net. The *action* then mimics the Petri net firing rule.

Example 1. Figure 3 presents a simple Petri net and its equivalent counter system. For readability purposes, the name of a variable is the same as the name of its associated place.

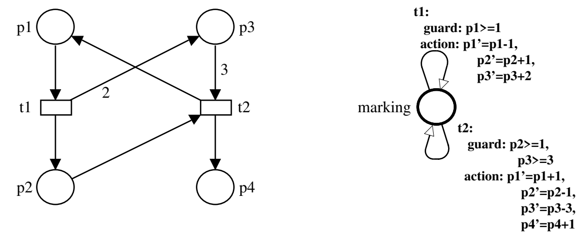

Fig. 3. A Petri net and its equivalent counter system

Many extensions of Petri nets can be modeled with counter systems, e.g.:

- *inhibitor arcs* check, in the transition guard, that the place contains *exactly* a given marking (or is empty) ;
- *read arcs* check, in the transition guard, that the place contains *at least* a given number of tokens, but does not remove these when firing;
- *reset arcs* set, in the transition action, the variable associated with the place to 0 ;
- *transfer arcs* set the variable associated with the output place to the value of the variable associated with the input place, and then set the latter to 0.

In our translation from PNML to counter systems, we handle normal, inhibitor, read and reset arcs (but not transfer arcs which are not yet included in PNML).

4.2 Implementation

The translation from Petri nets to counter systems is rather cumbersome. Hence, we wrote an automatic procedure to do so. To have a perennial procedure, we decided to consider Petri net files written in basic PNML (Petri Net Markup Language) [BCvH+03,PNM], and translate them into a FAST file. The translation is performed using an XSLT [Cla99] stylesheet.

The stylesheet analyses a PNML file provided as input and generates the equivalent counter system as output, which can then be used by FAST. As we will see, the model of the system is completely translated from the PNML file, whereas a standard strategy is created, including reachability set generation, but should be modified by the user so as to specify the properties to check.

First, the names of the places in the net are declared as variables of the counter system. To do so, a template is applied to all places:

```
var </xsl:text>
  <xsl:apply-templates select="place|pnml:place" mode="vars">
    <xsl:with-param name="tot" select="count(place|pnml:place)"/>
  </xsl:apply-templates> <xsl:text>;</xsl:text>
  <xsl:text>
```

The template returns the name of the place if any, or its identifier otherwise:

```
<xsl:template match="place|pnml:place" mode="vars">
  <xsl:param name="tot" select="0"/>
  <xsl:value-of
    select="./name/value|./pnml:inscription/pnml:value"/>
  <xsl:choose>
    <xsl:when test="position()!=$tot">
      <xsl:text>, </xsl:text>
    </xsl:when>
  </xsl:choose>
</xsl:template>
```

A single state of the counter system, representing a net marking, is declared:

```
  <xsl:text>

states marking;

</xsl:text>
```

Then, each transition is handled by a template, detailed hereafter.

```
  <xsl:apply-templates select="transition|pnml:transition"/>
```

Once the transitions are modelled in the counter system, the model is complete. The last part of this higher-level processing generates the standard strategy for the net to be analysed:

```
  <xsl:text>

strategy strat {
        setMaxState(2000);
        setMaxAcc(100);

        Region init := {</xsl:text>
  <xsl:apply-templates
    select="place/initialMarking|pnml:place/initialMarking"/>
  <xsl:text>state=marking};

        Transitions trans := {</xsl:text>
  <xsl:apply-templates
    select="transition/name|pnml:transition/name">
    <xsl:with-param
      name="last" select="/pnml/net/transition[last()]/@id"/>
  </xsl:apply-templates>
  <xsl:text>};

        Region reach := post*(init, trans, 1);
}
  </xsl:text>
```

Two limits are fixed: the maximum number of states for the symbolic representation of a region, and the maximum number of accelerations to be tried. When these limits are reached, FAST increments the cycle length used for accelerations. The numbers set here are average values obtained after several tests of the tool. The automatic translation sets the initial marking of the net as initial region of the counter system. However, it might be worthwhile for the user to modify these values, e.g. to have parameterized results, as in the example of section 4.3. The set `trans` of transitions to be taken into account for accelerations contains all transitions of the net. Finally, the reachability set is computed and stored in a region `reach`, using forward reachability. This can also be changed by the user, e.g. to use backwards reachability [3]. Region `reach` can afterwards be used for checking properties, as in the example of section 4.3. The properties must be specified by the user.

We will now detail how the core of the model, i.e. the transitions, is translated. A transition in FAST counter systems is described by its name and four fields:

- `from` is the source state of the transition ;
- `to` is the destination state of the transition ;
- `guard` is a boolean formula on the variables which must evaluate to true for the transition to be enabled ;
- `action` describes the evolution of the values of variables when the transition is fired. This is done by means of formulae where a primed variable name (i.e. `thevar'` for variable `thevar`) indicates the new value of the variable (after firing the transition), while the non-primed name (e.g. `thevar`) denotes the value before firing the transition.

The source and destination nodes of every transition are the unique state, therefore the `from` and `to` fields are all equal to `marking`.

The main part of the XSLT stylesheet consists in writing guards and actions. Table 1 reviews the formulae of guards and actions for each type of arc. The arcs connect a place `p` and a transition `t`. They may have a value `n` in the PNML description, otherwise the default value in table 1 is used.

Arc type	Default value	Guard	Action
normal `p`→`t`	1	`p>=n`	`p'=p-n`
normal `t`→`p`	1	—	`p'=p+n`
inhibitor	0	`p=n`	—
read	1	`p>=n`	—
reset	0	—	`p=0`

Table 1. Different types of arcs: corresponding guards and actions

A template is applied to all arcs in order to construct the guard expression.

```
<xsl:apply-templates select="../arc|../pnml:arc" mode="guard">
  <xsl:with-param name="transit" select="$ident"/>
  <xsl:with-param name="lastg" select="$lastg"/>
</xsl:apply-templates>
```

It tests the arc type, on which the guard expression depends, for each arc having the transition as a target. If it is a reset arc, nothing is done, if it is an inhibitor arc, an equality test is generated, otherwise a greater than or equal test is produced.

[3] Which is known to terminate when the starting set is downwardclosed.

```
<xsl:if test="@target=$transit">
  <xsl:variable name="arctype" select="./type/text"/>
  <xsl:choose>
    <xsl:when test="$arctype='reset'"/>
    <xsl:when test="$arctype='inhibitor'">
      <xsl:variable name="source" select="@source"/>
      <xsl:for-each select="../place">
        <xsl:if test="@id=$source">
          <xsl:value-of
            select="./name/value|./pnml:inscription/pnml:value"/>
        </xsl:if>
      </xsl:for-each>
      <xsl:text>=</xsl:text>
      <xsl:call-template name="arcvalue"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="source" select="@source"/>
      <xsl:for-each select="../place">
        <xsl:if test="@id=$source">
          <xsl:value-of
            select="./name/value|./pnml:inscription/pnml:value"/>
        </xsl:if>
      </xsl:for-each>
      <xsl:text>>=</xsl:text>
      <xsl:call-template name="arcvalue"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:if>
```

The template `arcvalue` is used, and returns the value attached to the arc if any, and the default value otherwise.

Finally, the action is generated. To do so, a template is applied to each place. This allows for computing the total value to be added to or subtracted from the current place marking.

```
<xsl:apply-templates
  select="../place|../pnml:place" mode="connected">
  <xsl:with-param name="firstplace" select="$firstplace"/>
  <xsl:with-param name="trans" select="$ident"/>
</xsl:apply-templates>
```

If the place is connected to a reset arc, its marking becomes 0 :

```
<xsl:variable name="existsreset"
  select="../arc[@target=$trans and @source=$place and
    ./type/text='reset']|../pnml:arc[@target=$trans and
    @source=$place and ./type/text='reset']"/>
```

```
<xsl:choose>
  <xsl:when test="$existsreset!=''">
    <xsl:value-of select="$placename"/>
    <xsl:text>'=0</xsl:text>
  </xsl:when>
```

Otherwise, the sum of values of normal arcs from the transition to the place is computed, as well as the sum of values of normal arcs from the place to the transition.

```
  <xsl:otherwise>
    <xsl:variable name="placeplus">
      <xsl:apply-templates
        select="../arc[@target=$place and @source=$trans]|
          ../pnml:arc[@target=$place and @source=$trans]"
        mode="plus">
        <xsl:with-param name="diff" select="0"/>
      </xsl:apply-templates>
    </xsl:variable>
    <xsl:variable name="placeminus" default="0">
      <xsl:apply-templates
        select="../arc[@target=$trans and @source=$place]|
          ../pnml:arc[@target=$trans and @source=$place]"
        mode="minus">
        <xsl:with-param name="diff" select="0"/>
      </xsl:apply-templates>
    </xsl:variable>
```

The differences between these two sums is the modification to be applied to the marking. Hence, the action can be constructed.

```
    <xsl:if test="$placediff!=0">
      <xsl:value-of select="$placename"/>
      <xsl:text>'=</xsl:text>
      <xsl:value-of select="$placename"/>
      <xsl:if test="$placediff>0">
        <xsl:text>+</xsl:text>
      </xsl:if>
      <xsl:value-of select="$placediff"/>
    </xsl:if>
```

Note that, if the difference is null, no action is generated as the marking remains unchanged.

The other parts of the XSLT stylesheet are mainly tricks to put the appropriate connectors between sub-formulae of the guards and actions, and a ; to end them properly. Thus, we will not detail the code any further.

4.3 Example

Let us consider the Petri net in figure 4. The PNK [PNK] was used to generate the PNML file except for the special arc types which were added manually.

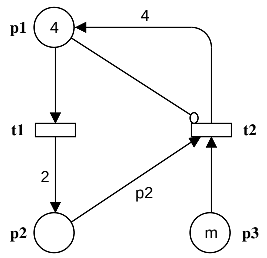

Fig. 4. A simple Petri net with inhibitor and reset arcs

The translation of this model into a FAST counter system is presented in figure 5.

```
/* This file was automatically generated from PNML */

model n1 {

var p1, p2, p3;

states marking;

transition t1 := {
 from   := marking;
 to     := marking;
 guard  := p1>=1;
 action := p1'=p1-1, p2'=p2+2;
};

transition t2 := {
 from   := marking;
 to     := marking;
 guard  := p3>=1 && p1=0;
 action := p1'=p1+4, p2'=0, p3'=p3-1;
};

}
```

Fig. 5. The FAST model

The strategy obtained via the XSLT stylesheet is presented in figure 6.

```
strategy strat {
        setMaxState(2000);
        setMaxAcc(100);

        Region init := {p1=4 && p2=0 && p3=m && state=marking};

        Transitions trans := {t1, t2};

        Region reach := post*(init, trans, 1);
}
```

Fig. 6. The generated strategy

We want to check that all reachable states satisfy $2.M(p1) + M(p2) = 2.4$, and that `p3` is bounded by its initial marking `m`. Hence, we add `m` as a variable in the model and modify the strategy, as shown in figure 7.

```
strategy strat {
        setMaxState(2000);
        setMaxAcc(100);

        Region init := {p1=4 && p2=0 && p3=m && state=marking};

        Transitions trans := {t1, t2};

        Region reach := post*(init, trans, 1);
        Region inv := {p1+p1+p2=8 && state=marking};
        Region maxp3 := {p3<=m && state=marking};
        if (subSet(reach,inv))
        then print("inv OK");
        else print("inv NOK");
        endif
        if (subSet(reach,maxp3))
        then print("bound OK");
        else print("bound NOK");
        endif
}
```

Fig. 7. The modified strategy

After running FAST, we conclude that both properties are satisfied, and in particular that parameter `m` is a bound for place `p3`.

5 Case Studies

FAST has been successfuly applied to several non-trivial Petri nets with a large or infinite state space. The results on files automatically generated from PNML are presented in figure 8. The models are available at [FAS], as well as other case studies. From the following table, we can see that FAST has similar performances on finite and infinite state systems. The number of variables and the length of loops to be accelerated are much more relevant factors for the computation time. The swimming pool example, which requires to acccelerate loops of length 4, shows that the heuritics used to find the cycles works well in practice. Finally all these examples highlight that acceleration, even if it uses only a semi-algorithm, is sound for practical automatic verification.

Case study	variables	transitions	time (s)	n. of accelerations	cycle length	n. of cycles
Bounded Petri Nets						
Producer/Consumer	5	3	0.26	1	1	4
Lamport ME	11	9	1.50	11	1	10
Dekker ME	22	22	13.10	36	1	23
RTP	9	12	1.25	8	1	13
Peterson ME	14	12	2.81	12	1	13
Reader/Writer	13	9	2.50	12	1	10
Petri Nets						
CSM	14	13	33.19	63	2	35
Multipoll	18	21	18.18	13	1	22
Kanban	16	16	6.53	22	1	17
Swimming Pool	9	6	191.13	13	4	47
Petri Nets with Inhibitor/Read Arcs						
Barber	8	12	1.07	8	1	13
FMS	22	20	102.24	23	2	46

Fig. 8. Results using an Intel Pentium 4, 2.4 Ghz with 768 Mbytes

6 Conclusion

Petri nets is a well-spread model to describe concurrent systems. However, the analysis often becomes intractable when the state space is large or even infinite.

We present a method to check complex properties on such Petri nets. First, Petri nets (with possibly inhibitor/read/reset arcs) in PNML format are automatically translated into counter systems. Then, the correctness of the counter system (thus of the Petri net) is checked with the FAST tool. FAST is a tool to analyze counter systems, among which Petri nets and some of their extensions. Hence, adopting PNML as a native format for FAST is out of scope. FAST allows to check parameterized reachability properties on infinite state spaces. Several non-trivial Petri nets with large or infinite state spaces have been analyzed following this method.

The contribution of this paper is twofold: on the one hand, it shows how the availability of a widely accepted interchange format can promote and ease the accessibility to new analysis tools, even those not originally developed for Petri nets, and on the other hand, it demonstrates the practical impact of the XML technology choice.

Future works include extending the automatic translation to more complex classes of nets. The verification of Petri nets properties with FAST requires the modification of the generated strategy. Further work on PNML, including expected properties, would allow to generate them automatically as well. For the moment, this task is not too tedious nor difficult as the names of places and transitions are easily tractable in the counter system. Improving FAST capabilities is also an issue: two promising approaches are the use of better symbolic representations and more efficient acceleration techniques.

References

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2000.

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1996.

[BCvH+03] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

[BF99] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, Aug. 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 1999.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Université de Liège, 1998.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94), Stanford, CA, USA, June 1994*, volume 2725 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.

[Cla99] J. Clark. XSL Transformations (XSLT) Version 1.0. URL `http://www.w3.org/TR/XSLT/xslt.html`, 1999.

[CPN] DESIGN/CPN *online*. `http://www.daimi.au.dk/designCPN`.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 352–359. IEEE Comp. Soc. Press, 1999.

[FAS] FAST *homepage*. `http://www.lsv.ens-cachan.fr/fast/`.

[FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, Dec. 2002*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[Ler03] J. Leroux. The affine hull of a binary automaton is computable in polynomial time. In *Proc. 5th Int. Workshop on Verification of Infinite State Systems (INFINITY'2003), Marseille, France, Sep. 2003*, Electronic Notes in Theor. Comp. Sci. Elsevier Science, 2003.

[PNK] Petri Net Kernel. `http://www.informatik.hu-berlin.de/top/pnk/`.

[PNM] Petri Net Markup Language. `http://www.informatik.hu-berlin.de/top/pnml/`.

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.