ECOLE NORMALE SUPERIEURE

Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic

Laurent FRIBOURG
Hans OLSÉN

LIENS - 96 - 24

Département de Mathématiques et Informatique

CNRS URA 1327

Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic

Laurent FRIBOURG
Hans OLSÉN*

LIENS - 96 - 24

Décembre 1996

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 00 00

Adresse électronique : fribourg@dmi.ens.fr

*Department of Computer and Information Science
Linköping University
S-58183 Linköping, Sweden

Adresse électronique : hanol@ida.liu.se

Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic

Laurent Fribourg*

Ecole Normale Supérieure & CNRS
45 rue d'Ulm, 75005 Paris - France
fribourg@dmi.ens.fr

Hans Olsén

IDA, Linköping University
S-58183 Linköping - Sweden
hanol@ida.liu.se

Abstract. We present in this report a method mixing top-down and bottom-up computation features for characterizing the reachability sets of Petri nets within Presburger arithmetic. An application of our method is the automatic verification of safety properties of Petri nets with infinite reachability sets. Our method is basically top-down in the sense that it uses structural features of the net for selecting (by "decomposition") canonical paths that go from a generic marking to another one. However we show here how to enhance the procedure by propagating the specific values of the initial marking in a bottom-up manner and proceeding to some invariance test. The approach is illustrated on three significant examples.

1 Introduction

We are interested in this report in proving safety properties of infinite state systems. We will focus on Petri nets although our approach is applicable to other discrete models of concurrent systems such as automata with counters (see, e.g., [8]). There will be two sources of infinity for the state space of Petri nets that we will consider: the first one is the unboundedness of some places of the net; the second one comes from the fact that the initial marking of the net may contain parameters, thus representing an infinite family of markings. The safety property that we will consider, we will be of the form;

$$\forall \overline{x}\ (\overline{x} \in lfp \Rightarrow I(\overline{x}))$$

where $\overline{x}$ represents a marking, lfp represents the set of reachable markings of the Petri net, and $I(\overline{x})$ an arithmetic relation characteristic of the safety property to be proved. Our method consists in characterizing the reachability relation $\overline{x} \in lfp$ as a Presburger arithmetic formula (i.e. an arithmetic formula without $\times$). The objective of our work is therefore similar to the one of Hiraishi

* Part of this work was done while the author was visiting IASI-CNR (Roma), supported by HCM-Network Logic Program Synthesis and Transformation CHRX.CT.930414

[9]. However Hiraishi constructs this arithmetic characterization in a bottom-up manner by refining Karp-Miller's method for constructing coverability trees [12]. In contrast, we construct the arithmetic characterization of the set of reachable markings using a basic top-down method of decomposition [15, 7]. Nevertheless, as explained in the rest of this report, we will also integrate in the basic top-down procedure some forward (bottom-up) computation routines that will speed up the arithmetic construction by propagating the initial marking values and testing the intermediate formulas for invariance.

2 Preliminaries

We will reason in this report with a certain class of logic programs with constraints over domain $\mathcal{Z}$ [10], which are of the form:

$$\begin{array}{lll} & p(\overline{x}) & \leftarrow\ B(\overline{x}). \\ r_1: & p(\overline{x}+\overline{t}_{r_1}) & \leftarrow\ \overline{x} > \overline{a}_{r_1},\ p(\overline{x}). \\ & \vdots & \\ r_m: & p(\overline{x}+\overline{t}_{r_m}) & \leftarrow\ \overline{x} > \overline{a}_{r_m},\ p(\overline{x}). \end{array}$$

where $\overline{x}$ is a vector of variables ranging over $\mathcal{Z}^n$, for some n, $B(\overline{x})$ a linear integer relation (relation defined by a Presburger formula), $\overline{t}_{r_i} \in \mathcal{Z}^n$ is a vector of constants, and $\overline{a}_{r_i}$ is a vector of constants belonging to $(\mathcal{Z} \cup \{-\infty\})^n$. As usual, $z > -\infty$, $z \neq -\infty$ and $-\infty \pm z = z \pm (-\infty) = -\infty$ for any integer $z \in \mathcal{Z}$, and $-\infty \geq -\infty$. For any vectors $\overline{x}_1$ and $\overline{x}_2$, we define $\overline{x}_1 > \overline{x}_2$ (resp. $\overline{x}_1 \geq \overline{x}_2$) to hold, if and only if the inequalities hold componentwise. $\max(\overline{x}_1, \overline{x}_2)$ is the vector obtained by taking the maximum of $\overline{x}_1$ and $\overline{x}_2$ componentwise. (thus, $\max(\overline{x}_1, \overline{x}_2)$ is the least upper bound in the $\langle (\mathcal{Z} \cup \{-\infty\})^n, \geq \rangle$-lattice. The vector with all components $-\infty$ is the bottom element.). Since $z > -\infty$ holds for any $z \in \mathcal{Z}$, any constraint of the form $x > -\infty$, is simply considered as *true*.

One can see these programs as classical programs with counters expressed under a logic programming or Datalog form. These programs have thus the power of expressivity of Turing machines. Henceforth we will refer to this class of programs as *programs with $\mathcal{Z}$-counters*. In the next section, we will see how these programs naturally encode the *reachability problem* for Petri nets (with inhibitors).

We now introduce a convenient description of the forward (or bottom-up) execution of programs with $\mathcal{Z}$-counters.

A *clause*, r, is characterized by a pair $\langle \overline{t}_r, \overline{a}_r \rangle$, where $\overline{t}_r \in \mathcal{Z}^n$ and $\overline{a}_r \in (\mathcal{Z} \cup \{-\infty\})^n$. We say that r is *applicable* at a point $\overline{x} \in \mathcal{Z}^n$ iff $\overline{x} > \overline{a}_r$ holds. The result of applying the clause r at a point $\overline{x}$ is $\overline{x}r = \overline{x} + \overline{t}_r$. More generally, let $\Sigma = \{r_1, \ldots, r_m\}$. A sequence $w \in \Sigma^*$ is called a *path*, and is interpreted as a sequence of forward applications of the clauses (in a bottom-up manner). Given

some point $\overline{x}$, the point reached by applying the path w is denoted $\overline{x}w$. Formally: $\overline{x}w = \overline{x} + \overline{t}_w$, where $\overline{t}_w$ is defined by:

$$\begin{aligned} \overline{t}_\varepsilon &= \overline{0} \\ \overline{t}_{rw} &= \overline{t}_r + \overline{t}_w \end{aligned}$$

Note that the expression $\overline{x}w$ does not take the constraints in the bodies of the clauses into account. We say that a *path w is applicable* at a point $\overline{x}$, if all constraints along the path are satisfied, and we write $\overline{x} > \overline{a}_w$, where:

$$\begin{aligned} \overline{a}_\varepsilon &= -\overline{\infty} \\ \overline{a}_{rw} &= max(\overline{a}_r, \overline{a}_w - \overline{t}_r) \end{aligned}$$

The expression $\overline{x} > \overline{a}_w$ is said to be the *constraint* associated to path w at point $\overline{x}$. The definition of $\overline{a}_w$ is based on the observation that $\overline{x} > max(\overline{a}_r, \overline{a}_w - \overline{t}_r)$ iff $\overline{x} > \overline{a}_r \ \wedge \ \overline{x} + \overline{t}_r > \overline{a}_w$, which means that $\overline{x} > \overline{a}_{rw}$ holds iff $\overline{x}_i > \overline{a}_{r_i}$ holds for every point $\overline{x}_i$ along the path rw. That is, the constraints associated with the clauses are satisfied at every point along the path.

It is immediately seen that, for programs with $\mathcal{Z}$-counters, the constraint associated with a path, is of the same form as that of a clause of the original program. In general, with every path w, there is associated a clause $\langle w \rangle = \langle \overline{t}_w, \overline{a}_w \rangle$.
A point $\overline{x}'$ is *reachable* from a point $\overline{x}$ by a path w if $\overline{x}w = \overline{x}'$ and w is applicable at $\overline{x}$:

$$\overline{x} \xrightarrow{w} \overline{x}' \ \Leftrightarrow \ \overline{x}w = \overline{x}' \ \wedge \ \overline{x} > \overline{a}_w$$

A point $\overline{x}'$ is reachable from a point $\overline{x}$ by a language $L \subseteq \Sigma^*$ if there exists a path $w \in L$ such that $\overline{x}'$ is reachable from $\overline{x}$ by w:

$$\overline{x} \xrightarrow{L} \overline{x}' \ \Leftrightarrow \ \exists w \in L : \ \overline{x} \xrightarrow{w} \overline{x}'$$

We usually write $\overline{x} \xrightarrow{L_1} \overline{x}'' \xrightarrow{L_2} \overline{x}'$, instead of $\overline{x} \xrightarrow{L_1} \overline{x}'' \ \wedge \ \overline{x}'' \xrightarrow{L_2} \overline{x}'$. From the definitions above, we immediately get:

Proposition 1. *For any path $w \in \Sigma^*$ and any languages $L_1, L_2 \subseteq \Sigma^*$. We have:*

1. $\overline{x} \xrightarrow{L_1 L_2} \overline{x}' \ \Leftrightarrow \ \exists \ \overline{x}'' : \ \overline{x} \xrightarrow{L_1} \overline{x}'' \xrightarrow{L_2} \overline{x}'$
2. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow \ \exists \ k \geq 0 : \ \overline{x}' = \overline{x} + k \cdot \overline{t}_w \ \wedge \ \forall \ 0 \leq k' < k : \ \overline{x} + k' \cdot \overline{t}_w > \overline{a}_w$

Note, in the last equivalence, that if $k = 0$, then $\overline{x} = \overline{x}'$ and $\forall \ 0 \leq k' \leq k$: $\overline{x} + k' \cdot \overline{t}_w > \overline{a}_w$ is vacuously true. It is easy to see that, for $k > 0$, the universally quantified subexpression is equivalent to $\overline{x} + (k-1) \cdot \overline{t}_w^- > \overline{a}_w$ where $\overline{t}_w^-$ is the vector obtained from $\overline{t}_w$ by letting all nonnegative components be set to zero. Therefore, the whole equivalence becomes:

2′. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow \ \overline{x}' = \overline{x} \ \vee \ \exists \ k > 0 : \ \overline{x}' = \overline{x} + k \cdot \overline{t}_w \ \wedge \ \overline{x} + k \cdot \overline{t}_w^- > \overline{a}_w + \overline{t}_w^-$

As a consequence, given a path w, the relation $\overline{x} \xrightarrow{w^*} \overline{x}'$ is actually an *existentially* quantified formula of Presburger arithmetic having $\overline{x}$ and $\overline{x}'$ as free variables. More generally, define a *flat* language as: any language of the form $w_1^*....w_c^*$ where each w_i $(1 \leq i \leq c)$ is an element of Σ^*. By proposition 1 1, it follows that the relation $\overline{x} \xrightarrow{L} \overline{x}'$ for a flat language L, can be expressed as an existentially quantified formula of Presburger arithmetic, having $\overline{x}$ and $\overline{x}'$ as free variables. More precisely, the reachability relation $\overline{x} \xrightarrow{L} \overline{x}'$ is expressed as a disjunction of a number of matrix expressions of the form:

$$\exists\ \overline{k}_i :\ \overline{x}' = \overline{x} + C_i\overline{k}_i\ \wedge\ \overline{x} + D_i\overline{k}_i > \overline{e}_i$$

where C_i and D_i are matrices, and $\overline{e}_i$ some vector of constants. Such a formula can be simplified as a formula, say $\zeta_L(\overline{x}, \overline{x}')$, by elimination of the existentially quantified variables $\overline{k}_i$ through a Presburger decision procedure (see [13]).

Given a program with $B(\overline{x})$ as a base case and recursive clauses Σ, the least fixed-point of its immediate consequence operator (see [10][11]), which is also the least $\mathcal{Z}$-model, may be expressed as:

$$\text{lfp} = \{\ \overline{x}' \mid \exists \overline{x} :\ B(\overline{x})\ \wedge\ \overline{x} \xrightarrow{\Sigma^*} \overline{x}'\ \}$$

Our aim is to characterize the membership relation $\overline{y} \in$ lfp as a closed formula having $\overline{y}$ as a free variable. In order to achieve this, our approach here is to find a flat language $L \subseteq \Sigma^*$, such that the following equivalence holds: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{L} \overline{x}'$. An arithmetic characterization of $\overline{y} \in$ lfp is then: $\exists \overline{x}\ B(\overline{x}) \wedge \zeta_L(\overline{x}, \overline{y})$. Such a formula can be in turn simplified as, say $\xi_{B,L}(\overline{y})$ by elimination of $\overline{x}$ through a decision procedure for Presburger arithmetic.
Given a formula $\xi(\overline{x})$ and a path w, we call *w-closure* of $\xi(\overline{x})$, a formula $\xi'(\overline{x})$ that characterizes arithmetically the set $\{x' \mid \xi(\overline{x}) \wedge \overline{x} \xrightarrow{w^*} \overline{x}'\}$.
We say that a path w lets *invariant* a formula ξ if $\xi(\overline{x}) \wedge \overline{x} \xrightarrow{w} \overline{x}'$ implies $\xi(\overline{x}')$, for all $\overline{x}, \overline{x}'$.
In the following, given a formula ξ and a language L, we will often abbreviate an expression of the form $\xi(\overline{x}) \wedge \overline{x} \xrightarrow{L} \overline{x}'$ as $\xi(\overline{x}) \xrightarrow{L} \overline{x}'$. The w-closure of a formula ξ will be accordingly denoted as $\xi \xrightarrow{w^*}$.

3 Encoding of the reachability problem of Petri Nets

Consider a Petri net with n places and m transitions. In this section, we show how to encode the reachability problem for Petri nets, via an n-ary predicate p defined by a program with $\mathcal{Z}$-counters. Each place of the Petri net will be encoded as an arithmetic variable x_j $(1 \leq j \leq n)$. A marking corresponds to a tuple $(v_1, ..., v_n)$ of n positive or null integers. (The value v_j represents the number of tokens contained in place x_j.) Each transition will be encoded as a recursive clause r_i $(1 \leq i \leq m)$. An atom of the form $p(v_1, ..., v_n)$ means that a marking $(v_1, ..., v_n)$ is reachable from the initial marking. The predicate p is defined as follows:

- The base clause r_0 is of the form:

$$p(x_1, ..., x_n) \leftarrow x_1 = v_1^0, ..., x_n = v_n^0.$$

  where $\overline{v}^0 = \langle v_1^0, ..., v_n^0 \rangle$ denotes the initial marking.
- The clause r_i $(1 \leq i \leq m)$, coding for the i-th transition, is of the form:

$$p(x_1 + t_{i,1}, ..., x_n + t_{i,n}) \leftarrow \phi_i(x_1, ..., x_n), p(x_1, ..., x_n).$$

  where $t_{i,j}$ is the sum of the weights of the output arrows from transition i to place j, minus the sum of the weights of the input arrows from place j to transition i. $\phi_i(x_1, ..., x_n)$ is: $x_{j_1} > a_{j_1} \wedge ... \wedge x_{j_{c_i}} > a_{j_{c_i}} \wedge x_{k_1} = 0 \wedge ... \wedge x_{k_{d_i}} = 0$, when $x_{j_1}, ..., x_{j_{c_i}}$ are the input places, and $x_{k_1}, ..., x_{k_{d_i}}$ the inhibitors places. (The condition ϕ_i expresses that the i-th transition is enabled.)

Note that such a program does not belong to the class we consider due to the constraints of the form $x_{k_\alpha} = 0$ $(1 \leq \alpha \leq d_i)$. However by adding extra arguments, say x'_{k_α} $(1 \leq \alpha \leq d_i)$, which are initialized with 1 minus the initial value of x_{k_α}, and are incremented (resp. decremented) when x_{k_α} is decremented (resp. incremented), one can replace the constraint $x_{k_\alpha} = 0$ with $x'_{k_\alpha} > 0$. (Note that x'_{k_α} is, by construction, always equal to $1 - x_{k_\alpha}$, and may thus take negative values.)

The least fixed-point lfp associated with the program corresponds to the *reachability set* associated with the Petri net, i.e. the set of all the markings reachable from the initial marking. Sometimes it is interesting to reason generically with some *parametric initial markings*, i.e., initial markings where certain places are assigned parameters instead of constant values. This defines a family of Petri nets, which are obtained by replacing successively the parameters with all the possible nonnegative values. One can easily encode the reachability relation for a Petri net with a parametric initial marking via a program with $\mathcal{Z}$-counter by adding the initial marking parameters as extra arguments of the encoding predicate. In the case of a Petri net with an initial marking containing a tuple of parameters, say $\overline{q}$, our aim is to characterize the relation $\overline{y} \in$ lfp as an arithmetical formula $\xi(\overline{q}, \overline{y})$ having $\overline{q}$ and $\overline{y}$ as free variables. This will allow us to determine all the values of the parameters $\overline{q}$ for which a given safety property holds (see subsection "Swimming Pool").

Example 1. Consider the Petri net in figure 1. (This example is the "swimming-pool" net from M. Latteux, see [3, 6].) With the initial marking $x_1 = x_2 = x_3 = x_4 = x_5 = 0$, $x_6 = q_1$ and $x_7 = q_2$ for some nonnegative parameters q_1 and q_2, the task is to show that there exists a deadlock regardless of what q_1 and q_2 are. The program P encoding the reachability problem for this Petri Net is the following:

r_0 : $p(q_1, q_2, x_1, x_2, x_3, x_4, x_5, x_6, x_7) \leftarrow$
$x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = q_1, x_7 = q_2.$

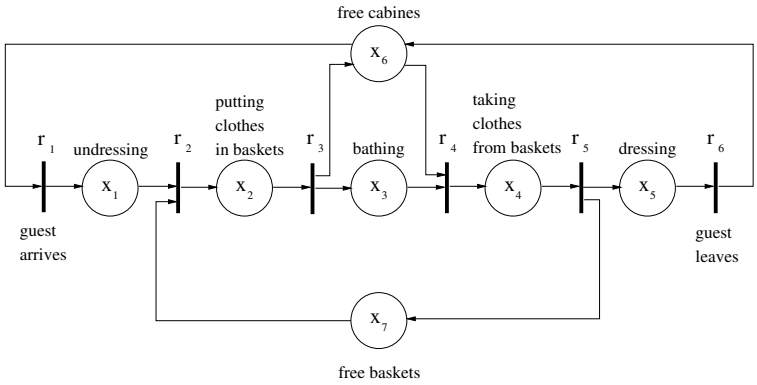

Figure 1

$$
\begin{aligned}
r_1: \quad & p(q_1,q_2,x_1+1,x_2,x_3,x_4,x_5,x_6-1,x_7) \leftarrow x_6>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).\\
r_2: \quad & p(q_1,q_2,x_1-1,x_2+1,x_3,x_4,x_5,x_6,x_7-1) \leftarrow x_1>0, x_7>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).\\
r_3: \quad & p(x_1,x_2-1,x_3+1,x_4,x_5,x_6+1,x_7,q_1,q_2) \leftarrow x_2>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).\\
r_4: \quad & p(q_1,q_2,x_1,x_2,x_3-1,x_4+1,x_5,x_6-1,x_7) \leftarrow x_3>0, x_6>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).\\
r_5: \quad & p(q_1,q_2,x_1,x_2,x_3,x_4-1,x_5+1,x_6,x_7+1) \leftarrow x_4>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).\\
r_6: \quad & p(q_1,q_2,x_1,x_2,x_3,x_4,x_5-1,x_6+1,x_7) \leftarrow x_5>0, \\
& \qquad\qquad p(q_1,q_2,x_1,x_2,x_3,x_4,x_5,x_6,x_7).
\end{aligned}
$$

4 Construction of Reachability Sets

Let us consider a program defined by a set of transitions $\Sigma_{original} : \{r_1, ..., r_m\}$. In order to characterize the relation $\overline{x} \xrightarrow{\{r_1,...,r_m\}^*} \overline{x}'$, we will construct a sequence $\{L_i\}_i$ of subsets of $\{r_1, ..., r_m\}^*$ which are "reachably-equivalent" to $\{r_1, ..., r_m\}^*$ in the sense that, for any $\overline{x}$ and $\overline{x}'$:

$$\overline{x} \xrightarrow{\{r_1,...,r_m\}^*} \overline{x}' \;\Leftrightarrow\; \overline{x} \xrightarrow{L_i} \overline{x}'$$

and such that the last language in the sequence is flat.
Such a flat language $L \subseteq \{r_1, ..., r_m\}^*$ will be generated by applying repeatedly a set of *decomposition* rules. Schematically, each decomposition rule, when applied to a set Σ, transforms it into a list Δ of the form $[\Sigma_1, \Sigma_2, \ldots, \Sigma_c]$, where Σ_i $(1 \leq i \leq c)$ denotes a program of "lower dimension" than Σ, and such that Σ^* is reachably-equivalent to the language $\Sigma_1^* \Sigma_2^* \ldots^* \Sigma_c^*$. The process of decomposition is iterated on list Δ: one element of Δ is selected, and the list resulting from its decomposition is inserted in place of it within Δ, thus generating a new sequence Δ'. The process is iterated until either:

- all the elements of the current list Δ are singletons of the form $\{w_1\}, \{w_2\}, \cdots, \{w_c\}$, which means that the associated language $w_1^* w_2^* ... w_c^*$ is flat (*termination with success*), or
- no decomposition rule applies onto the current list Δ (*termination with failure*).

Note that the process cannot loop forever because each decomposition rule transforms a program into a sequence of programs of "lower dimension" [7].
The number of rules of decomposition is 5. They are: stratification, monotonic transition, monotonic guard, cyclic postfusion and cyclic prefusion. They are tried in this order, and the first that succeeds is applied. (The rules of monotonic transition and cyclic postfusion are given in appendix 1; for details, see [7].)
When a flat language $L : \ w_1^* \ldots w_c^*$ has been generated, a decision procedure for Presburger arithmetic is invoked in order to construct formula $\xi_{B,L}$ (see section 2). Starting from the base case relation, the arithmetic decision procedure computes $\xi_{B,L}$ by extending the base case relation B with the successive w_i-closures $(1 \leq i \leq c)$. Formally, $\xi_{B,L}(\overline{x}')$ is defined to be $\xi_c(\overline{x}')$ where:

$$\begin{aligned} \xi_0(\overline{x}') &\equiv B(\overline{x}') \\ \xi_{i+1}(\overline{x}') &\equiv \exists \overline{x} : \ \xi_i(\overline{x}) \ \wedge \ \overline{x} \xrightarrow{w_{i+1}^*} \overline{x}' \end{aligned}$$

Actually a more efficient system is implemented by invoking earlier the arithmetic decision procedure, and starting to construct $\xi_{B,L}$ during the decomposition process, without waiting for the flat language L to be fully generated. This is explained in the next section.
It is sometimes interesting, besides, to keep track of the number of times k_i each w_i is repeated inside sequences of the form $w_1^* \ldots w_c^*$. We are therefore led to construct formulas of the form:

$$\begin{aligned} \xi_0(\overline{x}') &\equiv B(\overline{x}') \\ \xi_{i+1}(\overline{x}', \overline{k}_i, k_{i+1}) &\equiv \exists \overline{x} : \ \xi_i(\overline{x}, \overline{k}_i) \ \wedge \ \overline{x} \xrightarrow{w_{i+1}^{k_{i+1}}} \overline{x}' \end{aligned}$$

Such a construction is useful when one wishes to exhibit some "counter-example" path $w_1^{k_1} \cdots w_c^{k_c}$ which ends at a marking that violates the safety property under study (see 7.1).

5 General Description of the System

As illustrated in figure 2 the system consists of a decomposition procedure and a decision procedure for Presburger arithmetic.
We will represent the sequence Δ of decomposed languages as a list. Initially, Δ contains a single element: $\Sigma_{original}$. At each step, the *leftmost element* (head) of Δ is selected for further decomposition. The process can be schematized as

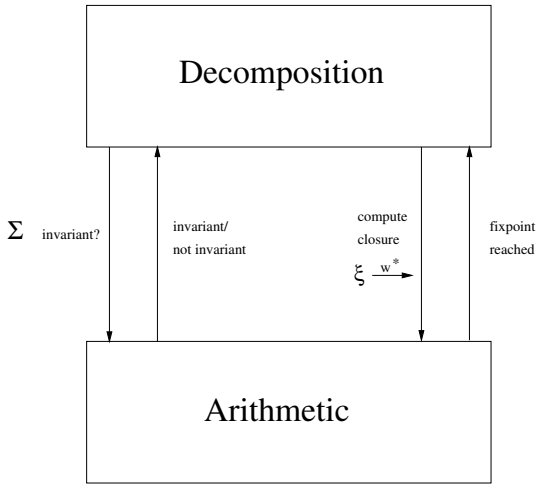

Figure 2

follows:

$$\overline{x} \xrightarrow{\Sigma^*_{original}} \overline{x}'$$
$$\Downarrow$$
$$\overline{x} \xrightarrow{\Sigma^*_{[1]}\Sigma^*_{[2]}\Sigma^*_{[3]}} \overline{x}'$$
$$\Downarrow$$
$$\overline{x} \xrightarrow{\Sigma^*_{[1,1]}\Sigma^*_{[1,2]}\Sigma^*_{[2]}\Sigma^*_{[3]}} \overline{x}'$$
$$\Downarrow$$
$$\overline{x} \xrightarrow{\Sigma^*_{[1,1,1]}\Sigma^*_{[1,1,2]}\Sigma^*_{[1,1,3]}\Sigma^*_{[1,2]}\Sigma^*_{[2]}\Sigma^*_{[3]}} \overline{x}'$$
$$\vdots$$

The system builds up a formula ξ, which will eventually characterize the least fixed-point. This formula is initialized with the base case relation B, and is extended by Σ-closure whenever the head Σ of Δ is a singleton. Before attempting any decomposition onto Σ, one checks whether it lets invariant by ξ (because there is no point in decomposing a set of transitions that will not yield anything new). The top loop of our procedure is thus as follows:

$\xi := B$;
$\Delta := [\Sigma_{\text{original}}]$;
while not empty(Δ) **do**
    $\Sigma :=$ head(Δ);
    $\Delta :=$ tail(Δ);
    if not invariant(ξ, Σ) **then**
        if singleton(Σ) **then**
            $\xi := \xi \xrightarrow{\Sigma^*}$
        else
            $\Delta :=$ decompose(Σ) $\otimes$ Δ
        fi

fi
od

where Δ is a list of sets of transitions, ξ is a Presburger formula, and $\otimes$ is append.

The arithmetic form $\xi_{B,L}$ of the least fixed-point is given by the exit value of ξ when executing the program . Henceforth, we will denote this exit formula by ξ_{final}. The corresponding flat language L is the sequence of closures of the singleton sets Σ popped up all along the execution of the program.
The invariance check before attempting decomposition is important since surprisingly often it happens that transition sets are discarded for which the decomposition would have failed at some point.

5.1 Invariance check

As mentioned, before attempting to decompose a set of transitions, we first check whether all the transitions in the top language Σ let the current arithmetic formula ξ, computed so far, invariant. If this is the case, the set is simply dropped and attention is moved to the next set. That is, before decomposing Σ, we check whether $\xi(\overline{x}) \xrightarrow{\Sigma} \overline{x}' \Rightarrow \xi(\overline{x}')$ holds (see section 2). This is *a priori* a computationally expensive test. However, by storing in a set $\Im$ those transitions that have been discovered to keep $\xi(\overline{x})$ invariant, a lot of redundant computations are avoided. Consider for example $\xi(\overline{x}) \xrightarrow{\{w_1,w_2,w_3\}^*} \overline{x}'$. Before trying to decompose $\{w_1, w_2, w_3\}^*$ we test invariance for each of the transitions. Assume that at least one of the three fails to let $\xi(\overline{x})$ invariant and that the decomposition rule of "monotonic transition" (see appendix 1) applies to w_2, say. At the next step we have to consider $\xi(\overline{x}) \xrightarrow{\{w_1,w_3\}^* w_2^* \{w_1,w_3\}^*} \overline{x}'$ and once again before decomposition of $\{w_1, w_3\}^*$ we test for invariance. But both w_1 and w_3 have already been tested, so the invariance check consists at this point in a table look up. Thus before testing invariance of a path w "the hard way", we first check whether it is among those in $\Im$.

On the other hand, when computing a w-closure of $\xi(\overline{x})$, the information in $\Im$ is *a priori* lost, and the new formula $\xi'(\overline{x})$ has a new set $\Im'$ of invariant transitions, which should be constructed. Here again, a lot of costly invariance tests can be saved by observing that a transition, say v, of $\Im$, which commute with w is guaranteed to be still in $\Im'$. Formally:

Proposition2. *Suppose:*

1. $\xi(\overline{x}) \xrightarrow{v} \overline{x}' \;\Rightarrow\; \xi(\overline{x}')$ *invariance of* w

2. $\xi'(\overline{x}') \;\equiv\; \exists \overline{x} :\; \xi(\overline{x}) \xrightarrow{w^*} \overline{x}'$ *w-closure*

3. $\overline{x} \xrightarrow{wv} \overline{x}' \;\Rightarrow\; \overline{x} \xrightarrow{vw} \overline{x}'$ *commutation*

Then invariance of v *is preserved:*

$$\xi'(\overline{x}) \xrightarrow{v} \overline{x}' \;\Rightarrow\; \xi'(\overline{x}')$$

Proof

$\xi'(\overline{x}) \xrightarrow{v} \overline{x}'$

$\Downarrow$ precondition 2: definition of $\xi'(\overline{x})$

$\exists \overline{x}''\ \xi(\overline{x}'') \xrightarrow{w^* v} \overline{x}'$

$\Downarrow$ precondition 3: commutation (by induction)

$\exists \overline{x}''\ \xi(\overline{x}'') \xrightarrow{v w^*} \overline{x}'$

$\Downarrow$ precondition 1: invariance (and proposition 1.1)

$\exists \overline{x}'''\ \xi(\overline{x}''') \xrightarrow{w^*} \overline{x}'$

$\Downarrow$ precondition 2: definition of $\xi'(\overline{x})$

$\xi'(\overline{x}')$

$\diamond$

As discussed in [7], .6, for programs with $\mathcal{Z}$-counters, the commutation check 2 of proposition 2 is computationally cheap.
The paths that fail the commutation check usually turn out to have lost their invariance.
Our experience from the examples we have considered is that at any point during a fixpoint computation about two thirds of the transitions of the original set $\Sigma_{original}$ are invariant. (At the end of the computation, naturally this ratio increases until becoming 100% when the fixed-point is reached.)

5.2 Failure of decomposition

So far in this section, we have assumed that the procedure of decomposition always succeeds. This may not be the case. In case of failure (i.e., when no rule of decomposition applies to the current set Σ), our strategy consists to remove some transitions from Σ according to some heuristics (essentially, random choice) until some decomposition rule applies or Σ becomes a singleton. This removal endangers the completeness of the finally generated formula ξ_{final} in the sense that it may not correspond any longer to a fixed-point. In such a case (i.e., when a transition of the original language $\Sigma_{original}$ does not let invariant ξ_{final}) the system detects it, and the whole procedure of fixed-point computation restarts except that ξ_{final} is taken as a new base case formula in place of B. This process is iterated until a fixed-point is actually reached. (There is no guarantee that such a fixed-point will be reached as the process may loop forever.) An example of such a process with transition removal and restarting, is given in section 7.3.

5.3 State Explosion

Our underlying decomposition strategy allows to alleviate (also for finite state systems) the problem of state explosion that immediately occurs with naive methods based on exhaustive state space exploration. This is because, among all the paths that go from the initial marking to a given reachable marking, the decomposition strategy leads to the selection of a reduced number of path "representatives" (see [15]). This selectivity in the choice of paths is reinforced by the interaction with the arithmetic module which, itself, discards quantities of invariant subsets of transitions. Naturally, even if our method allows us to treat automatically some examples that are usually done by hand (see section 7), we also have to face quickly with a state explosion problem. This is particularly sensitive in our case when one deals with Petri nets having more than one parameter in their initial markings. A solution for overcoming the problem is sometimes to reduce the original Petri net into a simpler net through transformation rules, as those of Berthelot [1], which preserve basic safety properties (e.g., deadlock-freeness, boundedness). An example of such a preliminary net transformation is given in section 7.1.

6 Arithmetic Module

The decision procedure for Presburger arithmetic that we have implemented is Boudet-Comon's algorithm [2]. It has turned out to be very well suited for our needs.
Given a system of equations and inequations, the Boudet-Comon algorithm generates a finite state automaton recognising the language of all solutions written as strings of binary digits.
This algorithm has nearly optimal worst case complexity and behaves according to our experience very well in practice. One of the advantages is its simplicity. Variable elimination, conjunction, disjunction, negation and inclusion are all achieved by standard automata theoretic methods such as projection, intersection, union, complement and emptiness testing. A great advantage of Boudet-Comon method is that, due to its simplicity and generality, it is easy to construct specialized programs for computing specific relations on its top, or to store information during its execution. We have exploited this feature for making easier the proof of general safety properties such as boundedness and detection of deadlock, as explained hereafter. A drawback of the method is that, from the automaton, there is no simple way to retrieve a closed form expression of the arithmetic relation.

Detecting unboundedness is achieved by investigating wether the reachability set is finite or infinite which is done efficiently done investigating the loops in the automaton representing the arithmetic relation.

A deadlock in a Petri net may defined by:

$$\text{deadlock}(\overline{q}, \overline{x}) \equiv \xi_{final}(\overline{q}, \overline{x}) \wedge \text{no\_transition\_enabled}(\overline{x})$$

where no_transition_enabled is specified as:

$$\text{no\_transition\_enabled}(\overline{x}) \;\equiv\; \forall r_i \in \Sigma_{original} : \neg\phi_i(\overline{x})$$

Explicitly defining no_transition_enabled($\overline{x}$) as above and then computing the automaton and intersecting with the fixed-point ξ_{final}, is not so efficient. We have therefore implemented a simple deadlock detector that directly computes ("on the fly") the automaton defining the relation deadlock($\overline{q}, \overline{x}$) according to the definition above throughout the construction of ξ_{final}.

7 Experimental Results

In this section we present some experimental data from three Petri nets: the two first ones have parametrized initial markings while the third one is unbounded. We generate for each of them the reachability sets under the form of a Boudet-Comon automaton, and are then able to prove for them various properties. The implementation has been written in SICSTUS-Prolog by the second author. It is around 4000 lines long, and runs on SPARC-10.

With each example, we give some statistics: number of variables (places) of the net, number of parameters of the initial marking, number of transitions of the net, finiteness of the reachability set, number of closures the flat computed language is composed, run time as well as two tables. The columns of the first table are to be interpreted as:

4. S Stratification.
5. MT Monotonic Transition.
6. MG Monotonic Guard.
7. PoF Cyclic Post Fusion.
8. PrF Cyclic Pre Fusion.
9. ND No Decomposition rule applies.

and the number in the column is the number of times the corresponding decomposition rule was applied.

The table IT (Invariant Transition set) two rows are presented: The top row is the number of transitions in the set, and the bottom row is the number of times a set of this size was discarded.

7.1 Swimming Pool

This example comes from M. Latteux (see, e.g.,[6]). Consider the Petri net in figure 1. With the initial marking $x_1 = x_2 = x_3 = x_4 = x_5 = 0$, $x_6 = q_1$ and $x_7 = q_2$ for some parameters q_1 and q_2, the task is to show that there exists a deadlock whatever the values of q_1 and q_2 are. (The proof is done by hand in [6].)

Our implementation does not succeed in computing the fixpoint since the automaton representing the reachability set grows too large (SICSTUS aborts after having generated 2500 states when determinizing an automaton having 386 states with 252 transition from each). So we apply our method not on the original net, but on a reduced version obtained by applying manually Berthelot's postfusion rule (fusing r_2 and r_3, and *eliminating* x_2) [1]. The reduced net is represented at figure 3. For any values of q_1 and q_2, the reduced net is guaranteed to be deadlock-free iff the original one is.

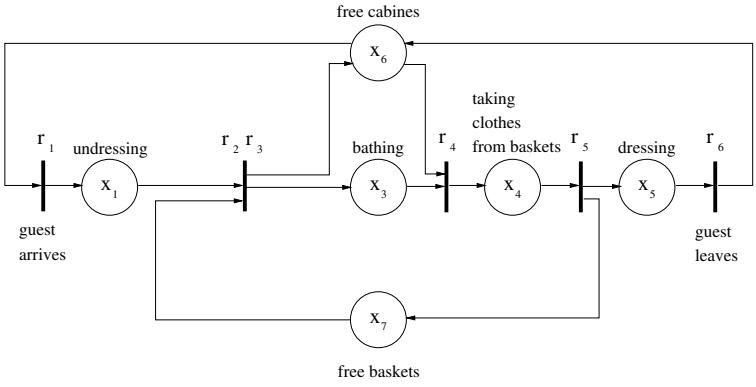

Figure 3

Computing the parametric reachability set we have the following statistics:

Net places:	6
Net parameters:	2
Net transitions:	5
Reachability set:	infinite
Computed language:	10 closures
Runtime:	10422 seconds (3.9 hours)

S	MT	MG	PoF	PrF	tot	ND
3	0	2	4	1	10	0

IT:

no. transitions	1	2	3	tot
no. disposals	8	1	3	12

The flat computed language L is:
$r_1^*(r_2r_3r_1)^*(r_2r_3)^*r_4^*r_5^*(r_2r_3)^*(r_4r_5r_2r_3)^*(r_4r_5)^*r_1^*r_4^*$
For the reduced swimming pool net of figure 3 the relation deadlock$(q_1, q_2, \overline{x})$ is computed in 12.27 seconds, and:

$$\forall q_1, q_2, : \ \exists \overline{x} : \ \text{deadlock}(q_1, q_2, \overline{x})$$

(that is, for any q_1 and q_2 there is a deadlock) is verified in 0.02 seconds.

For every couple of values q_1 and q_2, the system can compute path vectors in order to characterize paths leading to a deadlock. This gives (among others) paths of the form $r_1^{q_1}(r_2r_3r_1)^{q_2}$. (The paths leading to a deadlock found *by hand* in [6] are: $(r_1r_2r_3)^{q_2}r_1^{q_1}$.)

7.2 Manufacturing System

This example is taken from [5] (cf. [16]). Consider the Petri of figure 4. It models

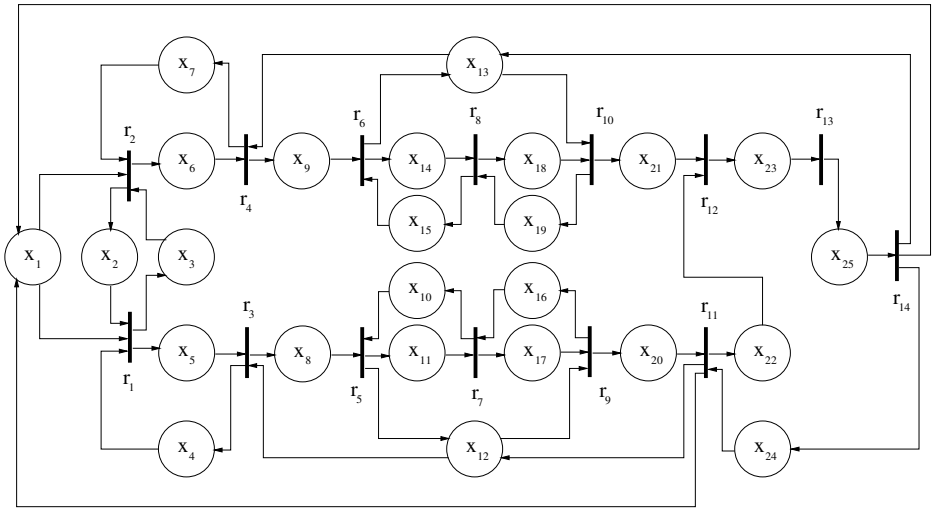

Figure 4

an automated manufacturing system with four machines, two robots, two buffers (x_{10} and x_{15}) and an assembly cell. The initial marking is: $x_1 = q$ for some non-negative parameter q, $x_2 = x_4 = x_7 = x_{12} = x_{13} = x_{16} = x_{19} = x_{24} = 1$, $x_{10} = x_{15} = 3$ (thus, the buffers have capacity 3). All other places are empty (that is, all other variables are 0). The task is to discover for which values of q the system may end up in deadlock. (In [16], deadlock-freeness is shown only for $1 \leq q \leq 4$. In [5], deadlock-freeness is proved using some mixed integer programming techniques for $1 \leq q \leq 8$; A path leading to a deadlock is also generated for $q = 9$.)

Computing the reachability set, we get the following statistics:

Net places:	25
Net parameters:	1
Net transitions:	14
Reachability set:	infinite
Computed language:	60 closures
Runtime:	23396 seconds (6.5 hours)

S	MT	MG	PoF	PrF	tot	ND
0	2	61	37	14	114	0

IT:

no. transitions	1	2	3	4	5	6	7	8	9	10	11	12	tot
no. disposals	42	24	18	24	19	13	8	17	2	1	2	1	171

The explicit form of the flat computed language L is given in appendix 2.

The relation deadlock$(q, \overline{x})$ is computed in 11.9 seconds. Define the relation:

$$\text{live}(q) \;\equiv\; \neg\exists\overline{x} : \; \text{deadlock}(q, \overline{x})$$

Thus live(q) is the set of parameters for which there is no deadlock in the system, and it is computed in 0.09 seconds. In 0.01 seconds, the cardinality of live(q) is found to be 8. Since there is no known way of retrieving from the automata a closed form expression of the relation, we simply enumerate them and get: $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

Thus, we have a fully automated proof that the system is deadlock free for all initial markings (as given above) for which $1 \leq q \leq 8$, and that for all other values a deadlock exists (note that from deadlock$(q, \overline{x})$, any deadlock for any q may be retrieved, as well as a path to any of them).

To prove that system is bounded for any q amounts to verifying:

$$\forall q \; \exists\overline{b} \; \forall\overline{x} : \; \xi_{final}(q, \overline{x}) \;\Rightarrow\; \overline{x} \leq \overline{b}$$

Our system is too naively implemented to prove this formula as stated, so instead we verify something stronger:

$$\text{subsystem}_1(x_2, x_3, \ldots, x_{25}) \;\equiv\; \exists q, x_1 : \; \xi_{final}(q, x_1, x_2, \ldots, x_{25})$$

The projection subsystem$_1(x_2, x_3, \ldots, x_{25})$ is computed in 38.74 seconds and is shown to be finite (it has 2144 elements) in 1.22 seconds. This shows that all the places but x_1 are bounded. Secondly we compute

$$\text{subsystem}_2(q, x_1) \;\equiv\; \exists x_2, x_3, \ldots, x_{25} : \; \xi_{final}(q, x_1, x_2, \ldots, x_{25})$$

in 7.89 seconds and prove:

$$\text{subsystem}_2(x_1, q) \;\Rightarrow\; x_1 \leq q$$

in 0.03 seconds. Therefore the system is bounded for all values of q.

7.3 Alternating Bit Protocol

This example is taken from [4] where all the correctness proofs are done by hand. Consider the alternating bit protocol of figure 5. Note that the system has 8

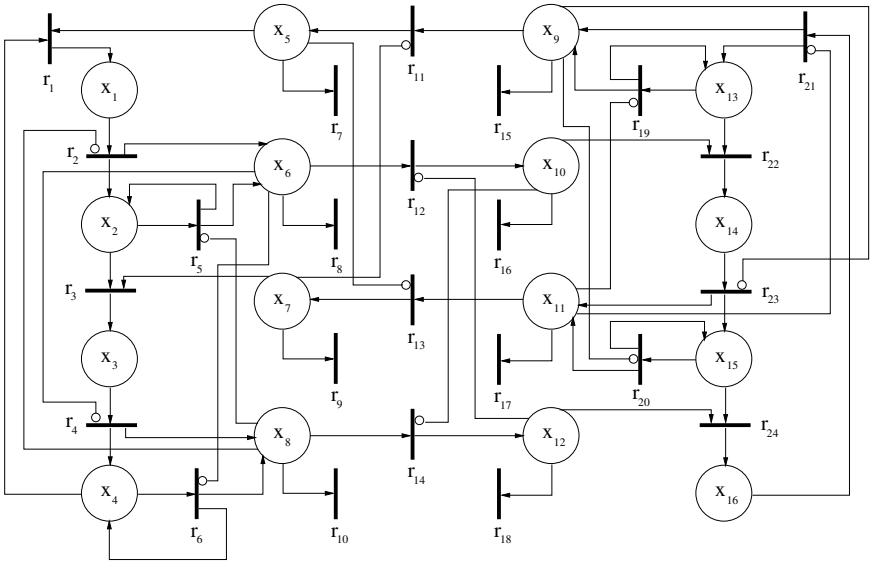

Figure 5

inhibitor places, which are simulated with 8 extra variables. Since there are 16 places in the net, we get a decomposition problem with 24 variables.
In this example, when computing the reachability set, the decomposition process fails several times. So the system drops some transition chosen according to some simple heuristic (basically random choice) The fixpoint is then computed with the following statistics:

Net places:	24
Net parameters:	0
Nettransitions:	24
Reachability set:	infinite
Computed language:	32 closures
Runtime:	127.60 seconds

S	MT	MG	PoF	PrF	tot	ND
12	24	48	26	0	110	6

IT:

no. transitions	1	2	4	5	6	7	8	9	10	11	12	13
no. disposals	23	7	3	4	6	4	9	7	6	3	6	3

We give here a typical mode of computation of the least fixed-point, made of three rounds of decomposition. During the first round, the decomposition process

fails several times, which yields the removal (by random choice) of r_5, r_{18} and r_{11}. When the decomposition ends, r_5 and r_{11} are noninvariants. The language is $L_1 = r_2^* r_8^* r_{12}^* r_{22}^* r_{23}^* r_{20}^* r_{17}^* r_{19}^* r_{22}^* r_{13}^* r_3^* r_4^* r_6^* r_{14}^* r_{10}^* r_{24}^* r_{21}^* r_{19}^* r_{15}^*$ and was computed in 33.32 seconds. We restart the process with the reachability set computed so far as base relation. The decomposition procedure fails again. This time, one removes randomly transitions among those distinct from r_5 and r_{11} (because these transitions were noninvariant at the end of the first round): this yield the removal of r_7, r_{16} and r_{12}. Once again the decomposition ends without reaching the fixpoint. This time r_{12} is noninvariant. The language: $L_2 = r_{11}^* r_{23}^* r_{20}^* r_{17}^* r_1^* r_2^* r_5^* r_3^* r_8^*$ was computed in 47.27 seconds. Next round r_{17}, r_8, r_{15}, r_{23}, r_{19}, r_3 and r_{13} were dropped. The language: $L_3 = r_{12}^* r_4^* r_6^* r_{10}^*$ was computed in 47.01 seconds, and this time the fixpoint has been reached. The flat language corresponding to the fixpoint is therefore $L_1 L_2 L_3$. Such an experience with random removal of transitions and restarting has been conducted many times, and has always yielded the fixpoint (with various associated flat languages, and various number of 'rounds').

The correctness of the protocol is based on the following properties of the model:

$$\begin{array}{rl}
\text{i.} & \xi_{final}(\overline{x}) \ \wedge \ x_1 = 1 \ \Rightarrow \ x_{13} = 1 \ \wedge \ x_6 = x_{10} = x_{11} = x_7 = 0 \\
\text{ii.} & \xi_{final}(\overline{x}) \ \wedge \ x_3 = 1 \ \Rightarrow \ x_{15} = 1 \ \wedge \ x_8 = x_{12} = x_9 = x_5 = 0 \\
\text{iii.} & \xi_{final}(\overline{x}) \ \wedge \ x_{14} = 1 \ \Rightarrow \ x_2 = 1 \ \wedge \ x_{11} = x_7 = x_8 = x_{12} = 0 \\
\text{iv.} & \xi_{final}(\overline{x}) \ \wedge \ x_{16} = 1 \ \Rightarrow \ x_4 = 1 \ \wedge \ x_9 = x_5 = x_6 = x_{10} = 0
\end{array}$$

The four correctness properties were proved in 1.52 seconds each. In 8 seconds, the unbounded places x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11} and x_{12} were found. Note that they coincide exactly with the inhibitor places.

8 Final Remarks

We have illustrated on three significant (although relatively small) examples how our top-down method of decomposition enhanced by invariance tests and forward propagation of the initial values, allows to characterize arithmetically the infinite reachability sets of some Petri nets. We have also experienced successfully our procedure on classical examples with finite reachability sets such as dining-philosophers or Peterson's mutual exclusion algorithm. As observed by Hiraishi [9], such a kind of method is not universal because it is known that some Petri nets have reachability sets that are not characterizable in Presburger arithmetic. In practice the main problem that we have to deal with is the state explosion problem, which prevents the construction of Boudet-Comon's automaton. We have indicated a way to alleviate this problem by reducing the original net to a simpler one that retains its main safety properties, using Berthelot's transformations. Another promising approach is the *compositional* approach which splits the original net into several components and reduces the verification of the original safety property to the verification of local safety properties of the components (see, e.g., [14]).

References

1. G. Berthelot. "Transformations and Decompositions of Nets". *Advances in Petri Nets*, LNCS 254, Springer-Verlag, 1986, pp. 359-376.
2. A. Boudet and H. Comon, "Diophantine Equations, Presburger Arithmetic and Finite Automata", *Proc. STACS*, LNCS, Springer-Verlag, 1995, pp. 30–43.
3. G.W. Brams. *Réseaux de Petri: Théorie et Pratique*, Masson, Paris, 1983.
4. J.M. Couvreur and E. Paviot-Adet. "New Structural Invariants for Petri Nets Analysis". *Proc. Application and Theory of Petri Nets*, LNCS 815, Springer-Verlag, 1994.
5. F. Chu and X. Xie. *Deadlock Analysis of Petri Nets Using Siphons and Mathematical Programming*. Technical Report INRIA, 1996, 29 pp.
6. R. David and H. Alla. *Du Grafcet aux Réseaux de Petri*, Hermès, Paris, 1989.
7. L. Fribourg and H. Olsén. *A Decompositional Approach for Computing the Least Fixed-points of Datalog Programs with Z-counters*. Technical Report LIENS-96-12, Ecole Normale Supérieure, Paris, July 1996. (Available on http://www.dmi.ens.fr/dmi/preprints)
8. N. Halbwachs. "Delay Analysis in Synchronous Programs", *Proc. Computer Aided Verification*, LNCS 697, Springer-Verlag, 1993, pp. 333-346.
9. K. Hiraishi. "Reduced State Space Representation for Unbounded Vector State Spaces", *Proc. Application and Theory of Petri Nets*, LNCS 1091, Springer-Verlag, 1996, pp. 230-248.
10. J. Jaffar and J.L. Lassez. "Constraint Logic Programming", *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 111-119.
11. P. Kanellakis, G. Kuper and P. Revesz. "Constraint Query Languages". Internal Report, November 1990. (Short version in *Proc. 9th ACM Symp. on Principles of Database Systems*, Nashville, 1990, pp. 299-313).
12. R.M. Karp and R.E. Miller. "Parallel Program Schemata". *J. Computer and System Sciences: 3*, 1969, pp. 147-195.
13. M. Presburger. "Uber die Vollstandingen einer gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervotritt", *Comptes Rendus du premier Congrès des Mathématiciens des Pays Slaves*, Varszawa, 1929.
14. A. Valmari. "Compositional State Space Generation", *Advances in Petri Nets*, LNCS 674, Springer-Verlag, 1993, pp. 427-457.
15. H-C. Yen. "On the Regularity of Petri Net Languages". *Information and Computation 124*, 1996, pp. 168-181.
16. M.C. Zhou, F. Dicesare and A.A. Desrochers. "A Hybrid Methodology for Synthesis of Petri Net Models for Manufacturing Systems", *IEEE Trans. on Robotics and Automation*, vol. 8:3, 1992, pp. 350-361.

9 Appendix 1: Two Rules of Decomposition

The program will be represented under the form of an "incrementation matrix", T as follows: Each clause is associated with a row, and each argument place in the head of the clause is associated with a column of the matrix. The element in the j:th row and k:th column is the coefficient t^j_k. The j:th row of the matrix will be denoted T^j.

monotonic transition This decomposition rule applies to a matrix T which contains one row T_k whose coefficients are all nonnegative (or all nonpositive). Let us suppose that all the coefficients are nonnegative (the nonpositive case is analogous). The fact that coefficients $t_{k,i}$ are positive (for all i) means that an application of clause r_k, will not decrease the value of any variable, so any clause whose guard is satisfied before r_k, will remain applicable after. This is in particular true for r_k itself, so once r_k becomes applicable for the first time, it can be applied any number of times. ¿From this it follows that any sequence of clauses w of Σ can be reordered so that all the applications of r_k be gathered together. Formally, we have the property:

$$\forall \overline{x}, \overline{x'} : \ \overline{x} \xrightarrow{\Sigma^*} \overline{x'} \ \Leftrightarrow \ \overline{x} \xrightarrow{(\Sigma - \{r_k\})^* r_k^* (\Sigma - \{r_k\})^*} \overline{x =}'$$

cyclic postfusion This decomposition rule applies to a matrix T whose last l rows (possibly after reordering of rules), T_{l+1} to T_n (for some $0 \leq l \leq n$) satisfies:

- the k-th coefficient $t_{j,k}$ of T_j is equal to -1 for $l+1 \leq j \leq n$.
- the line T_{l+j} is made of coefficients all positive or null, besides the k-th one.
- the constraints of the clauses corresponding to lines $l+1$ to n are all of the form $x_k > 0$.
- the k-th column of T is made of positive or null coefficients, for the rows 1 to l.
- x_k does not occur in any constraint of the clauses corresponding to lines 1 to l.

The matrix T is thus of the following form:

$$\begin{array}{lll}
\bullet \ldots \bullet \ + \ \bullet \ldots \bullet & \cdots & : r_1 \\
\vdots & & \\
\bullet \ldots \bullet \ + \ \bullet \ldots \bullet & \cdots & : r_l \\
+ \ldots + \ -1 \ + \ldots + & x_k > 0 & : r_{l+1} \\
\vdots & & \\
+ \ldots + \ -1 \ + \ldots + & x_k > 0 & : r_n \\
\quad x_k & &
\end{array}$$

We will abbreviate as b_i (where $1 \leq i \leq n$) the values of the coefficients $t_{i,k}$ of column k. ($b_1, \ldots, b_l$ are positive or null, and $b_{l+1}, \ldots, b_n$ is -1.) Here again, the fact that $t_{j,i}$, $l+1 \leq j \leq n$, are positive (for all i except k) means that an application of a clause r_j, will not decrease the value of any variable other than x_k, so any clause, $r_{j'}$, $1 \leq j' \leq l$, whose guard is satisfied before r_j, will remain applicable after. Thus, r_j may be applied as soon as its guard becomes satisfied. But now, a clause r_j does not indefinitely apply immediately after its first application: the application of a clause r_j ceases when the k-th coordinate of the current tuple $\overline{x}$ becomes null. Then another clause r_i (for some $1 \leq i \leq l$) must be applied. The k-th coordinate of the newly generated tuple is then equal to b_i. If b_i is strictly positive, then one of the clauses r_j, $(l+1 \leq j \leq n)$ can be applied again a number of times equal to b_i until x_k becomes null again. This shows that any sequence w of Σ can be reordered into a sequence whose core is made of repeated "patterns" of the form $(r_i(r_{l+1} + \ldots + r_n)^{b_i})$, Note also that these "patterns" let x_k invariant,

and are applied when $x_k = 0$. Such patterns are also called "cyclic sequences" in the field of Petri nets. Formally, we have the following property:

$$\forall \overline{x}, \overline{x}' : \ \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Leftrightarrow \ \overline{x} \xrightarrow{(r_{l+1}+\ldots+r_n)^* exp (r_1+\ldots+r_l)^*} \overline{x}'$$

where exp is: $(\mu_1 + \ldots + \mu_l)^* \mu'$.
Here μ_i $(1 \leq i \leq l)$ stands for a set of clauses equal to:

- r_i if $b_i = 0$, or
- $r_i (r_{l+1} + \ldots + r_n)^{b_i}$, if $b_i > 0$.

The expression μ' denotes a set of "subpatterns". That is, an expression of the form:

- $\{\varepsilon\} \cup \bigcup_{1 \leq i \leq l} \bigcup_{0<c<b_i} r_i (r_{l+1} + \ldots + r_n)^c$

10 Appendix 2: Flat Language for the Manufacturing System

The flat language computed by the system for the example of subsection 7.2 is:
$L = r_1^* r_2^* r_4^* r_6^* r_8^* r_{10}^* r_3^* r_1^* r_2^* r_4^* r_6^* r_{10}^* r_8^* r_5^* (r_3 r_5)^* (r_1 r_3 r_5)^*$
$r_2^* r_4^* r_6^* r_{10}^* r_8^* r_1^* r_2^* r_4^* r_6^* r_{10}^* r_8^* r_3^* r_1^* r_2^* r_4^* r_6^* r_{10}^* r_8^* r_7^* r_5^* r_3^* r_1^* r_2^* (r_9 r_7)^* r_9^* r_{11}^* r_3^* r_5^* r_2^*$
$r_1^* r_3^* r_5^* r_7^* (r_2 r_1)^* r_2^* r_4^* r_6^* r_{10}^* r_8^* r_{10}^* r_9^* r_7^* r_{12}^* r_{13}^*$

This article was processed using the LaTeX macro package with LLNCS style