Petri Nets, Flat Languages and Linear Arithmetic

Laurent Fribourg

LSV - ENS Cachan & CNRS – 61 av. Pdt. Wilson, 94235 Cachan - France
email: {fribourg}@lsv.ens-cachan.fr

Abstract. We present a method for characterizing the least fixed-points of a certain class of Datalog programs in Presburger arithmetic. The method consists in applying some rules of decomposition that transform general sets of computation paths into "flat" ones. We apply the method for expressing the reachability relation of Petri nets as a linear arithmetic formula in order to prove (or disprove) their safety properties.

1 Introduction

The problem of computing fixpoints for arithmetical programs has been investigated from the seventies in an imperative framework. A typical application was to check whether or not array bounds were violated. A pionneering work in this field is the work by Cousot-Halbwachs [8]. The subject has known a renewal of interest with the success of model checking methods in the nineties. Halbwachs has shown in particular how to verification of safety properties of concurrent systems using dataflow analysis [17]. However his techniques are interested in finding not the *least* fixpoint but rather an approximation of it using some techniques of Abstract Interpretation (convex hull, widening, ...). We explain here how to find the *least* fixed points for logic programs with having all their arguments incremented by the recursive clauses. The arguments of the programs can be seen as *counters*. By applying a clause from-right-to left (in a forward/bottom-up manner), one increments all the the arguments providing that the constraints of the clause body are satisfied. The problem of computing least fixed-points for such programs is closely related, as will be explained, to the problem of characterizing the set of the reachable markings ("reachability set") of Petri nets. One difference is that the variables of our programs take their values on $\mathbb{Z}$ instead of $\mathbb{N}$ as in the case of Petri nets. We will see however that some transformation rules by "decomposition" defined for Petri nets, such as Berthelot's post-fusion rule [3], still apply to programs with $\mathbb{Z}$-counters. The fact that we manipulate variables taking their values on $\mathbb{Z}$ rather than on $\mathbb{N}$ will allow us to encode in a simple way the important extension of Petri nets with *inhibitors*, which have the expressive power of Turing machines. The problem of computing least fixed-points for such programs amounts to solve the reachability problem for Petri nets with inhibitors, which is undecidable in general. This means that our decomposition method cannot always succeeds. When it succeeds, it yields a "flat language" of transitions that covers the whole set of reachable markings. The binary reachability relation using this flat language is then expressible in Presburger arithmetic. We are able in practice to treat a large number of significant examples, including Petri nets with parametric initial markings. We have also shown that our method allows us to treat the binary reachability problem for a special class of Petri nets, called BPP-nets [10].

The plan of this paper is as follows. In section 2 we give some preliminaries. Section 3 recalls some basic facts about Petri nets. Section 4 gives the basic rules of our decompositional method. Section 5 explains how to compile the reachability relation via a flat language into Presburger arithmetic. work. Section 6 explains how to use arithmetic over reals instead of integers in order to speed up the compilation, and describes some experiments.

2 Preliminaries

Our aim in this paper is to express the least fixed-point of a certain class of logic programs as a linear integer arithmetic expression (a Presburger formula). We consider programs of the form:

$$\begin{array}{lll}
 & p(x_1,\ldots,x_m) & \leftarrow B(x_1,\ldots,x_m). \\
r_1: & p(x_1+k_{1,1},\ldots,x_m+k_{1,m}) & \leftarrow x_{i_{1,1}} > a_{1,1},\ldots,x_{i_{1,m_1}} > a_{1,m_1}, \\
 & & \quad p(x_1,\ldots,x_m). \\
 & & \vdots \\
r_n: & p(x_1+k_{n,1},\ldots,x_m+k_{n,m}) & \leftarrow x_{i_{n,1}} > a_{n,1},\ldots,x_{i_{n,m_n}} > a_{n,m_n}, \\
 & & \quad p(x_1,\ldots,x_m).
\end{array}$$

where $B(x_1,\ldots,x_m)$ is a linear integer relation, $k_{i,j}, a_{i,l} \in \mathbb{Z}$ are integer constants and r_j is simply the name of the j:th recursive clause. We will usually denote by $\overline{x}$ the vector $\langle x_1,\ldots,x_m\rangle$, by $\overline{k}_{r_j}$ the vector $\langle k_{j,1},\ldots,k_{j,m}\rangle$ and by $\vartheta_{r_j}(\overline{x})$ the constraint $x_{i_{j,1}} > a_{j,1},\ldots,x_{i_{j,m_j}} > a_{j,m_j}$. Usually the constants $a_{i,l}$ are equal to zero.

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. In the following we will refer to this class of programs as *(Datalog) programs with $\mathbb{Z}$-counters*. The motivation for studying this class is mainly because syntactically it is restricted enough to allow simple decomposition rules to be formulated and since there are no terms other than integers one can work within an arithmetic context. At the same time these programs are expressive enough to be interesting.

We introduce a convenient description of the execution of programs with $\mathbb{Z}$-counters in a bottom-up manner. Let $\Sigma = \{r_1,\ldots,r_n\}$. A string $w \in \Sigma^*$ is called a *path*, and is interpreted as a sequence of 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: $\overline{x}w = \overline{x} + \overline{k}_w$, where $\overline{k}_w$ is defined by:

$$
\begin{aligned}
\overline{k}_\varepsilon &= \overline{0} \\
\overline{k}_{r_j w} &= \overline{k}_{r_j} + \overline{k}_w
\end{aligned}
$$

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 $\vartheta_w(\overline{x})$. Formally:

$$
\begin{aligned}
\vartheta_\varepsilon(\overline{x}) &\Leftrightarrow \text{true} \\
\vartheta_{r_j w}(\overline{x}) &\Leftrightarrow \vartheta_{r_j}(\overline{x}) \ \wedge \ \vartheta_w(\overline{x}r_j)
\end{aligned}
$$

The expression $\vartheta_w(\overline{x})$ is said to be the *constraint* associated to path w at point $\overline{x}$. It is easily seen that $\vartheta_w(\overline{x})$ can be put under the form $x_{i_1} > a'_1, \ldots, x_{i_{m'}} > a'_{m'}$. As an example, consider the two clauses:

$$
\begin{array}{lll}
r_3: & p(x_2+1, x_3-1, x_4, x_5+1, x_6, x_7) & \leftarrow \quad x_3 > 0, \ \ p(x_2, \ldots, x_7). \\
r_5: & p(x_2, x_3, x_4, x_5-1, x_6+1, x_7) & \leftarrow \quad x_5 > 0, \ \ p(x_2, \ldots, x_7).
\end{array}
$$

The constraint $\vartheta_{r_3 r_5}(\overline{x})$ associated with $r_3 r_5$ is $\vartheta_{r_3}(\overline{x}) \wedge \vartheta_{r_5}(\overline{x}r_3)$, that is $x_3 > 0 \wedge x_5 + 1 > 0$, i.e.: $x_3 > 0 \wedge x_5 > -1$.

A point $\overline{x}'$ is *reachable* from $\overline{x}$ by a path w if $\overline{x}w = \overline{x}'$ and w is applicable at $\overline{x}$:

$$
\overline{x} \xrightarrow{w} \overline{x}' \ \Leftrightarrow \ \overline{x}w = \overline{x}' \ \wedge \ \vartheta_w(\overline{x})
$$

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 and any languages $L_1, L_2 \subseteq \Sigma^*$. We have:*

1. $\overline{x} \xrightarrow{L_1+L_2} \overline{x}' \ \Leftrightarrow \ \overline{x} \xrightarrow{L_1} \overline{x}' \ \vee \ \overline{x} \xrightarrow{L_2} \overline{x}'$
2. $\overline{x} \xrightarrow{L_1 L_2} \overline{x}' \ \Leftrightarrow \ \exists \ \overline{x}'' : \ \overline{x} \xrightarrow{L_1} \overline{x}'' \xrightarrow{L_2} \overline{x}'$
3. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow \ \exists \ \mu \geq 0 : \ \overline{x}' = \overline{x} + \mu \cdot \overline{k}_w \ \wedge \ \forall \ 0 \leq \mu' < \mu : \ \vartheta_w(\overline{x} + \mu' \cdot \overline{k}_w)$

In part 3 of the proposition, when $\mu = 0$, we have $\overline{x} = \overline{x}'$, and the expression $\forall \ 0 \leq \mu' < \mu : \ \vartheta_w(\overline{x} + \mu' \cdot \overline{k}_w)$ is vacuously true. Actually, the universally quantified variable μ' can always be eliminated from this expression. This is because the constraints $\vartheta_w(\overline{x})$ are necessarily of the form $\overline{x} \geq \overline{a}_w$, where $\overline{x} \geq \overline{a}_w$ denotes a conjunction of the form $x_{i_1} > a'_{1,w}, \ldots, x_{i_{m'}} > a'_{m',w}$. Thus, $\vartheta_w(\overline{x} + \mu' \cdot \overline{k}_w)$ is $\overline{x} + \mu' \cdot \overline{k}_w \geq \overline{a}_w$. It is easy to see that, for $\mu > 0$, the universally quantified expression $\forall\ 0 \leq \mu' < \mu :\ \overline{x} + \mu' \cdot \overline{k}_w > \overline{a}_w$ is equivalent to $\overline{x} + (\mu - 1) \cdot \overline{k}_w^{\,-} > \overline{a}_w$ where $\overline{k}_w^{\,-}$ is the vector obtained from $\overline{k}_w$ by letting all nonnegative components be set to zero. For example if $\vartheta_w(\overline{x})$ is $x_1 > 0 \wedge x_2 > 0$, and k_w is $\langle 2, -3 \rangle$, then $\forall\ 0 \leq \mu' < \mu :\ \overline{x} + \mu' \cdot \overline{k}_w > \overline{a}_w$ is

$$\begin{array}{ll} & \forall\ 0 \leq \mu' < \mu :\ x_1 + \mu' \cdot 2 > 0 \quad \wedge \quad x_2 + \mu' \cdot (-3) > 0 \\ \Leftrightarrow & x_1 + (\mu - 1) \cdot 0 > 0 \quad \wedge \quad x_2 + (\mu - 1) \cdot (-3) > 0 \\ \Leftrightarrow & x_1 > 0 \quad \wedge \quad x_2 - 3\mu + 3 > 0. \end{array}$$

As a consequence, from part 3 of the proposition, it follows that, given a finite sequence of transitions 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 suppose that L is a language of the form $L_1 ... L_s$ where L_i is either a finite language or a language of the form w_i^*, then, by proposition 1, it follows that the relation $\overline{x} \xrightarrow{L} \overline{x}'$ can be expressed as an existentially quantified formula of Presburger arithmetic having $\overline{x}$ and $\overline{x'}$ as free variables. We call such a language L a *flat* language (because Kleene's star operator '*' applies only to strings w). Given a program with $B(\overline{x})$ as a base case and recursive clauses labelled by Σ, the least fixed-point of its immediate consequence operator (see [20,21]), denoted $post^*_\Sigma(B)$, may be expressed as:

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

This set can be seen as the set of *iterated successors* of points of B via Σ. Alternatively, given a formula F characterizing a set of final markings, the set of *iterated predecessors* of F via Σ is defined as:

$$pre^*_\Sigma(F) = \{\ \overline{x} \mid \exists \overline{x}' :\ F(\overline{x}') \ \wedge\ \overline{x} \xrightarrow{\Sigma^*} \overline{x}'\ \}$$

Our aim is to characterize the membership relation $\overline{y} \in post^*_\Sigma(B)$ (or $\overline{y} \in pre^*_\Sigma(F)$) as an arithmetic formula having $\overline{y}$ as a free variable. This is useful for proving safety properties of the Petri net. For example if one wants to prove that no marking of F is reachable from an initial marking, it suffices to show $\overline{y} \in post^*_\Sigma(B) \Rightarrow \neg F(\overline{y})$ or alternatively $\overline{y} \in pre^*_\Sigma(F) \Rightarrow \neg B(\overline{y})$. For solving such a problem, it suffices actually to characterize the binary relation $\overline{x} \xrightarrow{\Sigma^*} \overline{x}'$ as an arithmetic formula having $\overline{x}$ and $\overline{x}'$ as free variables. In order to achieve this, our approach here is to find a flat language $L \subseteq \Sigma^*$, such that the following equivalence holds: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Leftrightarrow\ \overline{x} \xrightarrow{L} \overline{x}'$. This gives us an arithmetic characterization of the least fixed-point. The language L is constructed by making use of decomposition rules on paths. Such rules state that, if a path v links a point $\overline{x}$ to a point $\overline{x}'$ via Σ^*, then v can be reordered as a path w of the form $w = w_1 w_2 \cdots w_s$ such that $w_1, w_2, \cdots, w_s$ belong to some restricted languages. Such a "decompositional approach" is well known in Petri-net theory (see [27]).

3 The Reachability Problem for Petri Nets

3.1 Petri nets as programs with $\mathbb{Z}$-counters

There is a close connection between the class of programs with $\mathbb{Z}$-counters and Petri nets, and more precisely, between the computation of the least fixed-point of programs with $\mathbb{Z}$-counters and the "reachability problem" for Petri nets. Let us first give an informal explanation of what a Petri net is. (This is inspired from [10].) A Petri net is characterized by a set of places (drawn as circles), a set of transitions (drawn as bars), and for each transition τ, a set of weighted input-arcs going from a subset of places ("input-places") to τ, and a set of weighted output-arcs going from τ to a subset of places ("output-places"). A *marking* is a mapping of the set of places to the set $\mathbb{N}$ of nonnegative integers. The number assigned to a place represents the number of tokens contained by this place. A marking *enables* a transition τ if it assigns all the input places of τ with a number greater than or equal to the weight of the corresponding input-arc. If the transition is enabled, then it can be *fired*, and its firing leads to the successor marking, which is defined for every place as follows: the number of tokens specified by the weight of the corresponding input-arc is removed from each input place of the transition, and the number of tokens specified by the weight of the corresponding output place is added to each output place. (If a place is both an input and an output place, then its number of tokens is changed by the difference of weights between the corresponding output and input arcs.)

The reachability problem for a Petri net consists in characterizing the set of all the markings that are "reachable" from a given initial marking, that is the set of markings that can be produced by iteratively firing all the possible enabled transitions. Let us explain how the reachability problem of a Petri net with n transitions and m places can be encoded as a Datalog program with $\mathbb{Z}$-counters. Each place π_i of the Petri net is represented by a variable x_i, and its value encodes the number of tokens at that place. As a base case relation $B(\overline{x})$, one take the equation $\overline{x} = \overline{a}^0$ where $\overline{a}^0$ denotes the initial marking; each transition τ_j in the net is represented by a recursive clause r_j of the program as follows:

head constants: For each place π_i, the constant $k_{j,i}$ is equal to the weight of the output-arc going from τ_j to π_i, minus the weight of the input arc going from π_i to τ_j.

body constraints: For each input place π_i of transition τ_j, there is a constraint in the clause r_j of the form $x_i > a_{j,i} - 1$ where $a_{j,i}$ is equal to the weight of the input arc going out from π_i to τ_j. (No other constraints occur in the clause.)

Each clause of the program encodes the enabling condition of the corresponding Petri net transition. The above program therefore encodes the reachability problem for the considered Petri net: a tuple $\overline{y}$ belongs to the least fixed-point of the program iff it corresponds to a marking reachable from the initial one via the firing of a certain sequence of Petri net transitions. In other words the least fixed-point of the recursive program coincides with the set of the reachable markings ("reachability set") of the Petri net.

Note that the class of Datalog programs with $\mathbb{Z}$-counters is more general than the class of above programs encoding the reachability problem for Petri nets. From a syntactical point of view, the difference is that, with programs encoding the reachability problem, all the variables take their values on the domain $\mathbb{N}$ of non-negative integers while the domain for programs with $\mathbb{Z}$-counters is $\mathbb{Z}$. From a theoretical point of view, programs with $\mathbb{Z}$-counters have the power of Turing machines while (programs coding for the reachability problem of) Petri nets have not.

3.2 0-tests

There are many extensions to the Petri-net formalism, one of which allows *inhibitors* or 0-tests. In such extensions, the transitions may be conditioned by the fact that some input place contains 0 token. This test is materialized by the existence of an "inhibitor-arc" (represented as circle-headed arcs) from the place to the transition. Petri-nets with inhibitors are naturally encoded as Datalog programs with $\mathbb{Z}$-counters by adding a constraint $x_i = 0$ in the body of clause r_j whenever there is an inhibitor arc from place i to transition j. Within our framework where the variables of the program can take *negative* values, it is easy to simulate 0-tests. We encode inhibitor arcs by replacing a constraint $x_j = 0$ by $x'_j > 0$ where x'_j is a newly introduced variable. This new variable x'_j is to be equal to $1-x_j$. The variable x'_j is introduced as a new argument into p. Its initial value ${a'_j}^0$ is set to $1 - a^0_j$, where a^0_j denotes the initial value of x_j. Within each recursive clause r_i of the program, the new argument x'_j is incremented by $-k_{i,j}$ (where $k_{i,j}$ denotes the value the variable x_j is incremented by r_i). Formally, if we denote the newly defined predicate by p', we have in the least $\mathbb{Z}$-model of the union of the programs defining p and p': $p(\overline{x}) \Leftrightarrow \exists x'_j\ p'(\overline{x}, x'_j)$.

3.3 Parametric initial markings

It is often 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 positive or null values. One can easily encode the reachability relation for a Petri net with a parametric initial marking via a program with $\mathbb{Z}$-counter by adding the initial marking parameters as extra arguments of the encoding predicate. For the sake of notation simplicity however, we will not make such extra predicate arguments appear explicitly in the following. (The parameters will just appear in the base clause associated with the initial marking.) In the case of a Petri net with an initial marking containing a tuple of parameters, say $\overline{q}$, our aim will be to characterize the relation $\overline{y} \in post^*_{\Sigma}(B)$ as an arithmetical formula having $\overline{y}$ and $\overline{q}$ as free variables. We now illustrate the encoding of Petri-nets with inhibitors and parametric initial markings by an example.

Example 2. We consider here a Petri net implementing a simple readers-writers protocol. (This is inspired from [2], p.17.) This Petri net has six places encoded by the variables $x_2, x_3, x_4, x_5, x_6, x_7$ and six transitions encoded by the recursive clauses $r_1, r_2, r_3, r_4, r_5, r_6$. (It will be clear later on why the enumeration of places x_i starts with $i = 2$.) Place x_5 represents the number of idle processes. Place x_6 (resp. x_7) the number of candidates for reading (resp. writing). Place x_4 (resp. x_3) represents the number of current readers (resp. writers). Place x_2 is a semaphore for guaranteeing mutual exclusion of readers and writers. Only one inhibitor arc exists in the net, connecting x_4 to r_1. The Petri net is represented on figure 1. (The weights of the arcs are always equal to 1, and do not appear explicitly on the figure.) Only two places are initially marked: x_2 and x_5. The latter contains a parametric number of tokens, defined by the parameter q, while the former contains one token. The program P encoding this Petri-net is the following:

$$
\begin{array}{lll}
 & p(x_2, x_3, x_4, x_5, x_6, x_7) & \leftarrow x_2 = 1, x_3 = 0, x_4 = 0, x_5 = q, \\
 & & \quad q \geq 0, x_6 = 0, x_7 = 0. \\
r_1 : & p(x_2 - 1, x_3 + 1, x_4, x_5, x_6, x_7 - 1) & \leftarrow x_2 > 0, x_7 > 0, x_4 = 0, \\
 & & \quad p(x_2, \ldots, x_7). \\
r_2 : & p(x_2, x_3, x_4 + 1, x_5, x_6 - 1, x_7) & \leftarrow x_2 > 0, x_6 > 0, \\
 & & \quad p(x_2, \ldots, x_7). \\
r_3 : & p(x_2 + 1, x_3 - 1, x_4, x_5 + 1, x_6, x_7) & \leftarrow x_3 > 0, \\
 & & \quad p(x_2, \ldots, x_7). \\
r_4 : & p(x_2, x_3, x_4 - 1, x_5 + 1, x_6, x_7) & \leftarrow x_4 > 0, \\
 & & \quad p(x_2, \ldots, x_7). \\
r_5 : & p(x_2, x_3, x_4, x_5 - 1, x_6 + 1, x_7) & \leftarrow x_5 > 0, \\
 & & \quad p(x_2, \ldots, x_7). \\
r_6 : & p(x_2, x_3, x_4, x_5 - 1, x_6, x_7 + 1) & \leftarrow x_5 > 0, \\
 & & \quad p(x_2, \ldots, x_7).
\end{array}
$$

To replace the constraint $x_4 = 0$, we introduce the new variable x_1 and construct a new program P' defined in such a way that $x_1 = 1 - x_4$, holds in the least model of P', and replace $x_4 = 0$ by $x_1 > 0$ in clause r_1. We get:

$$
\begin{array}{lll}
 & p'(x_1, x_2, x_3, x_4, x_5, x_6, x_7) & \leftarrow x_1 = 1 - x_4, x_2 = 1, \\
 & & \quad x_3 = 0, x_4 = 0, x_5 = q, \\
 & & \quad q \geq 0, x_6 = 0, x_7 = 0. \\
r_1 : & p'(x_1, x_2 - 1, x_3 + 1, x_4, x_5, x_6, x_7 - 1) & \leftarrow x_2 > 0, x_7 > 0, x_1 > 0, \\
 & & \quad p'(x_1, \ldots, x_7). \\
r_2 : & p'(x_1 - 1, x_2, x_3, x_4 + 1, x_5, x_6 - 1, x_7) & \leftarrow x_2 > 0, x_6 > 0, \\
 & & \quad p'(x_1, \ldots, x_7). \\
r_3 : & p'(x_1, x_2 + 1, x_3 - 1, x_4, x_5 + 1, x_6, x_7) & \leftarrow x_3 > 0, \\
 & & \quad p'(x_1, \ldots, x_7). \\
r_4 : & p'(x_1 + 1, x_2, x_3, x_4 - 1, x_5 + 1, x_6, x_7) & \leftarrow x_4 > 0, \\
 & & \quad p'(x_1, \ldots, x_7). \\
r_5 : & p'(x_1, x_2, x_3, x_4, x_5 - 1, x_6 + 1, x_7) & \leftarrow x_5 > 0, \\
 & & \quad p'(x_1, \ldots, x_7). \\
r_6 : & p'(x_1, x_2, x_3, x_4, x_5 - 1, x_6, x_7 + 1) & \leftarrow x_5 > 0, \\
 & & \quad p'(x_1, \ldots, x_7).
\end{array}
$$

We have the following equivalence:

$$p(x_2, x_3, x_4, x_5, x_6, x_7) \;\Leftrightarrow\; \exists x_1 : \; p'(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

We would like to prove a mutual exclusion property for this protocol: reading

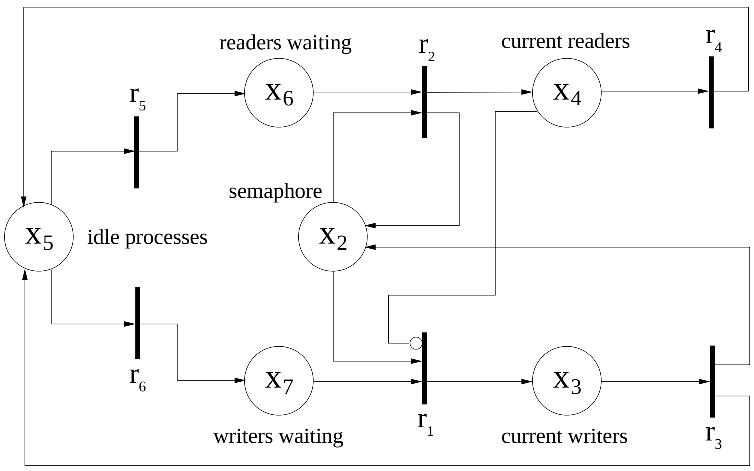

Fig. 1. A readers-writers protocol

and writing can never occur at the same time (i.e.: $x_3 = 0 \vee x_4 = 0$). Formally, we must prove that, in the least model of P':

$$p'(x_1, x_2, x_3, x_4, x_5, x_6, x_7) \;\Rightarrow\; (x_3 = 0 \vee x_4 = 0)$$

This will be done by computing $post^*_\Sigma(B)$ and showing $\overline{x} \in post^*_\Sigma(B) \Rightarrow x_3 = 0 \vee x_4 = 0$, where B and Σ are the base case and the set of clauses $r_1, \cdots, r_6$ of program P' respectively.

4 Construction of Least Fixed-points

The transformations we are going to present, only concern the recursive clauses. Since these clauses all have the same form (i.e. no reordering or sharing of variables, and all recursive calls are exactly the same) we will represent a program by an "incrementation matrix" whose j:th row is the vector $\overline{k_j}$ of coefficients of the j:th recursive clause of the program, and the constraints and the name of the clause are written to the right of the corresponding row of the matrix.

Example 3. The program P' of example 2 is represented by:

$$\begin{array}{rrrrrrrll}
0 & -1 & 1 & 0 & 0 & 0 & -1 & x_2 > 0, x_7 > 0, x_1 > 0 & : r_1 \\
-1 & 0 & 0 & 1 & 0 & -1 & 0 & x_2 > 0, x_6 > 0 & : r_2 \\
0 & 1 & -1 & 0 & 1 & 0 & 0 & x_3 > 0 & : r_3 \\
1 & 0 & 0 & -1 & 1 & 0 & 0 & x_4 > 0 & : r_4 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & x_5 > 0 & : r_5 \\
0 & 0 & 0 & 0 & -1 & 0 & 1 & x_5 > 0 & : r_6 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

Without loss of understanding, we will also call "program" the set Σ of the labels $r_1, .., r_n$ of the recursive clauses.

4.1 Decomposition rules

As explained before, the method we use to compute the reachability set consists in showing that any path can be reordered into some specific form of "lower dimension". In this paper we present in detail only two transformation rules. (The definition of some other rules is given besides in [14].) The rules are stated in the form: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Leftrightarrow\; \overline{x} \xrightarrow{L_1 L_2 \cdots L_s} \overline{x}'$. We say that Σ^* *decomposes as* $L_1 L_2 \cdots L_s$. Each languages L_i $(1 \leq i \leq s)$ denotes here either a finite language or a language of the form Σ_i^* where Σ_i is a label for a new program of lower dimension. Programs Σ_i are of lower dimension than the original program (labeled by Σ) by either containing a fewer number of recursive clauses, or by letting more variables invariant. (From a syntactic point of view, a variable is invariant when the corresponding column in the incrementation matrix is null.) See [14] for a formal definition of dimension.

The decomposition process either terminates successfully, thus giving a characterization of the reachability relation in Presburger arithmetic, or terminates with failure because no decomposition rule applies at some point. We consider two types of decomposition rules: *monotonic* and *cyclic* rules. Monotonic rules are based on the fact that some clauses of a program may be applied all at once at some point during a computation, while cyclic rules exploit that there is some fixed sequences of clause firings that can be repeated. We first present one monotonic decomposition rule, then one cyclic rule.

4.2 Monotonic decomposition rule

The first decomposition rule is called *monotonic clause*. This decomposition applies when there is a clause whose coefficients in the head are all nonegative or nonpositive. Thus, the monotonic clause is stated in two versions: one *increasing*, and one *decreasing*. For the purposes of this paper, we only state the increasing version (the decreasing one being symmetric). This rule applies to a program whose matrix is of the form:

$$\begin{array}{ccc|l}
 & \vdots & & \vartheta_{r_{\dots}} : r_{\dots} \\
+ & \dots & + & \vartheta_{r_\ell} : r_\ell \\
 & \vdots & & \vartheta_{r_{\dots}} : r_{\dots} \\
x_1 & & x_m &
\end{array}$$

This means that, in the program, we have $\forall j : k_{\ell,j} \geq 0$. In such a case, clause r_ℓ can be "prioritized" before all the rest of the clauses: given a path w starting at a point $\overline{x}$ where $\vartheta_{r_\ell}(\overline{x})$ holds, one can always reorder w so that all the clauses r_ℓ are applied first. Formally we have:

Proposition 4. *Let $r_\ell \in \Sigma$ be a clause such that $\forall j : k_{\ell,j} \geq 0$. Then:*

$$\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Leftrightarrow\; \overline{x} \xrightarrow{(\Sigma-\{r_\ell\})^* r_\ell^* (\Sigma-\{r_\ell\})^*} \overline{x}'$$

The proof is in [14]. Other monotonic decomposition rules are also given in [14].

4.3 Cyclic decomposition rule

The cyclic decomposition rule that we consider applies to matrices of the general form (after possible reordering among clauses $r_1, ..., r_n$):

$$\begin{array}{l}
\left.\begin{array}{ccccc|r}
\bullet \dots \bullet & + & \bullet \dots \bullet & & \cdots & : r_1 \\
 & \vdots & & & & \\
\bullet \dots \bullet & + & \bullet \dots \bullet & & \cdots & : r_\ell
\end{array}\right\} R' \\
\left.\begin{array}{ccccc|r}
+ \dots + & -1 & + \dots + & & x_j > 0 & : r_{\ell+1} \\
 & \vdots & & & & \\
+ \dots + & -1 & + \dots + & & x_j > 0 & : r_n \\
 & x_j & & & &
\end{array}\right\} R
\end{array}$$

where R and R' are sets of rules such that $\Sigma = R \uplus R'$, the constraints of all the clauses in R are exactly $x_j > 0$ and x_j does not occur in the constraints of any rule in R'. Formally this means
1. $\forall r_i \in R: \ k_{i,h} \geq 0$ for $h \neq j$
1'. $\forall r_i \in R': \ x_j$ does not occur in $\vartheta_{r_i}(\overline{x})$
2. $\forall r_i \in R': \ k_{i,j} \geq 0$
3. $\forall r_i \in R: \ k_{i,j} = -1$
4. $\forall r_i \in R: \ \vartheta_{r_i}(\overline{x}) \equiv x_j > 0$

Under conditions $1, 1', 2, 3, 4$, given a path w starting at a point where x_j is greater than 0, one can reorder w so that all the R-clauses are applied first (similarly to the situation of the monotonic transformation), but now such a priority of application for the R-clauses must end at some point: this is because, here, the coefficients $k_{i,j}$ $(\ell+1 \leq i \leq n)$ are not positive or null, but equal to -1. So the value of x_j decreases at each application of an R-clause until x_j becomes null. At this stage, no R-clause is applicable, and an R'-clause r_i $(1 \leq i \leq l)$ must be applied. The j-th coordinate of the newly generated tuple is then equal to $k_{i,j}$. If $k_{i,j}$ is strictly positive, then any of the "highest priority" R-clauses can be applied again a number of times equal to $k_{i,j}$ until x_j becomes null again. This shows that any path w of Σ^* can be reordered into a path whose core is made of repeated "cyclic sequences" of the form $r_i w$ with $w \in R^{k_{i,j}}$. (As usual, the expression R^k denotes the set of paths in R^* of length k.) Note that these "cyclic sequences" let x_j invariant, and are applied when $x_j = 0$. To summarize, the strategy of application of the clauses here is to apply R-clauses in priority, whenever they are applicable (i.e., when $x_j > 0$), until x_j becomes null.

Remarks Actually, requirements 1 and $1'$ that all the coefficients $k_{i,h}$ should be nonnegative (for $h \neq j$), and x_j should not occur in the R'-constraints, are unnecessarily strong. It is enough that, under condition $x_j > 0$, rules of R "commute" with those of R' in the following sense:

$$x_j > 0 \ \wedge \ \overline{x} \xrightarrow{R'R} \overline{x}' \ \Rightarrow \ \overline{x} \xrightarrow{RR'} \overline{x}' \ .$$

Requirement 4 can be also relaxed: a similar decomposition holds when the constraints of the R-clauses are not atomic (i.e., not equal to $x_j > 0$) but contain other guards (i.e., when $\vartheta_{r_i}(\overline{x}) \ \Rightarrow \ x_j > 0$).

Before stating formally the cyclic decomposition rule, we introduce and briefly comment on some notation used in the formal statement of the rule. The expression $r_i R^k$ denotes the set $\{r_i w \mid w \in R^k\}$. The expression $R^{0<*<k}$ denotes the set of paths w in R^* of length greater than 0 and less than k. If $r_i R^k$ represents a set of cyclic sequences, the expression $r_i R^{0<*<k}$ thus represents the set of *prefixes* of such sequences. (The prefix reduced to r_i is discarded by the notation, and appears in the rule statement as an element of R'.) The language $(\bigcup_{r_i \in R'} r_i R^{k_{i,j}})^*$ also appears in the rule statement. The program associated with this language is made of recursive clauses of the form: $p(\overline{x}+\overline{k}_{r_i w}) \leftarrow \vartheta_{r_i w}(\overline{x}), p(\overline{x})$., where w is in $R^{k_{i,j}}$. The dimension of such a program is less than the dimension of Σ because

it lets one more variable, viz. x_j, invariant (The x_j column in the corresponding incrementation matrix is null.)

Proposition 5. *Let $R, R' \subseteq \Sigma$ be sets of (labels of) clauses such that $\Sigma = R \uplus R'$, and let x_j be a variable such that:*

1. $x_j > 0 \;\wedge\; \overline{x} \xrightarrow{R'R} \overline{x}' \;\Rightarrow\; \overline{x} \xrightarrow{RR'} \overline{x}'$
2. $\forall r_i \in R' : \; k_{i,j} \geq 0$
3. $\forall r_i \in R : \; k_{i,j} = -1$
4. $\forall r_i \in R : \; \vartheta_{r_i}(\overline{x}) \;\Rightarrow\; x_j > 0$

Then we have

$$\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Leftrightarrow$$

$$\begin{array}{l} \overline{x} \xrightarrow{R'^*R^*R'^*} \overline{x}' \\ \qquad\qquad \vee \\ \exists \overline{x}'' : \; \overline{x} \xrightarrow{R'^*R^*} \overline{x}'' \xrightarrow{\left(\bigcup_{r_i \in R'} r_i R^{k_{i,j}}\right)^* \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right) R'^*} \overline{x}' \;\wedge \\ \qquad x''_j = 0 \end{array}$$

where x_j is let invariant by all the paths in $\left(\bigcup_{r_i \in R'} r_i R^{k_{i,j}}\right)^*$.

The proof is in [14]. Proposition provides us with a decomposition rule that reduces the reachability problem via Σ^* to several reachability problems via languages which are of lower dimensions. The sublanguages are R'^*, R^*, $(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}})$ and $(\bigcup_{r_i \in R'} r_i R^{k_{i,j}})^*$. Languages R'^* and R^* have fewer clauses than Σ^* (and at least as many variables kept invariant), so they are of lower dimension. The language $(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}})$ is finite. As already pointed out, the language $(\bigcup_{r_i \in R'} r_i R^{k_{i,j}})^*$ is of lower dimension than Σ^* because it leaves a new variable (viz., x_j) invariant.

Example 6. Consider the matrix in example 3, representing the program for the protocol of example 2. Let us in proposition 5 choose $R = \{r_5, r_6\}$ and $R' = \{r_1, r_2, r_3, r_4\}$, and let $x_j = x_5$. We see that this matrix conforms to the special case discussed above where the decomposition of proposition 5 is applicable. We have: $k_{1,5} = 0$, $k_{2,5} = 0$, $k_{3,5} = 1$ and $k_{4,5} = 1$. Thus, $r_1 R^{k_{1,5}} = r_1$, $r_2 R^{k_{2,5}} = r_2$, $r_3 R^{k_{3,5}} = r_3 r_5 + r_3 r_6$ and $r_4 R^{k_{4,5}} = r_4 r_5 + r_4 r_6$. Furthermore: $\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,5}} = \varepsilon + r_1 R^{0<*<0} + r_2 R^{0<*<0} + r_3 R^{0<*<1} + r_4 R^{0<*<1} = \varepsilon$. By proposition 5, we have:

$$\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5+r_6)^*} \overline{x}' \;\Leftrightarrow$$

$$\begin{array}{l} \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}' \\ \qquad\qquad \vee \\ \exists \overline{x}'' : \; \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*} \overline{x}'' \;\wedge \\ \qquad \overline{x}'' \xrightarrow{(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}' \wedge x''_5 = 0 \end{array}$$

and all the paths in $(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*$ keep $x_5 = 0$ invariant. The matrix M' of the program corresponding to the set of clauses $\{r_1, r_2, r_3, r_5, r_3r_6, r_4r_5, r_4r_6\}$ is shown below:

$$\begin{array}{ccccccclr}
0 & -1 & 1 & 0 & 0 & 0 & -1 & x_2 > 0, x_7 > 0, x_1 > 0 & : r_1 \\
-1 & 0 & 0 & 1 & 0 & -1 & 0 & x_2 > 0, x_6 > 0 & : r_2 \\
0 & 1 & -1 & 0 & 0 & 1 & 0 & x_3 > 0, x_5 > -1 & : r_3r_5 \\
0 & 1 & -1 & 0 & 0 & 0 & 1 & x_3 > 0, x_5 > -1 & : r_3r_6 \\
1 & 0 & 0 & -1 & 0 & 1 & 0 & x_4 > 0, x_5 > -1 & : r_4r_5 \\
1 & 0 & 0 & -1 & 0 & 0 & 1 & x_4 > 0, x_5 > -1 & : r_4r_6 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

Thus, $(r_1 + r_2 + r_3 + r_4)^*$ and $(r_5 + r_6)^*$ involves fewer clauses than the original program, while $(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*$ involves the same number of clauses but lets one more variable, viz. x_5, invariant. (The corresponding column in the incrementation matrix is null.)

4.4 Comparison with Berthelot's work

As can be seen in the example, in the matrix M' corresponding to the set of cyclic sequences, the constraint $x_5 > -1$ is systematically satisfied since it is applied, by proposition 3, to a point of coordinate $x_5 = 0$ and x_5 is kept invariant. So an obvious optimization, for the treatment of the matrix, will be to remove the null column as well as the guard $x_5 > -1$. In terms of Petri nets, this corresponds to remove the place x_5 and to perform the "fusion" of transitions r_2, r_3, r_4 (which have x_5 as an output place) and transitions r_5, r_6 (which have x_5 as an input place). The resulting Petri net is represented in figure 4.4. This kind of optimization can be done generally, under the preconditions of proposition 3. An analogous transformation of Petri nets is called *post-fusion* transformation by Berthelot in [3]: sequences r_3r_5, r_3r_6, r_4r_5 and r_4r_6 are called the "fused transitions". Our version of the cyclic decomposition can thus be seen as a variant of Berthelot's post-fusion rule. Berthelot also defined in [3] some other transformations like *pre-fusion*. It is possible to give in our framework a counterpart also for this transformation (see [14]). The point that should be stressed here is that our cyclic decomposition rules are more general than Berthelot's rules because they apply to general programs with $\mathbb{Z}$-counters where variables take their values on $\mathbb{Z}$ (instead of $\mathbb{N}$ as in the case of Petri nets). This allows us in particular to encode 0-tests as already seen. Besides, our goal is different from Berthelot's one. Here we are interested here in transformations of the language of rules that preserve the reachability relation. In contrast, Berthelot is interested in transformations of Petri nets that preserve special marking properties such as deadlock-freeness.

4.5 Application to BPP-Nets

Since programs with $\mathbb{Z}$-counters are Turing equivalent it is clear that the decomposition process cannot always succeed. Even for Petri nets it is known that nets

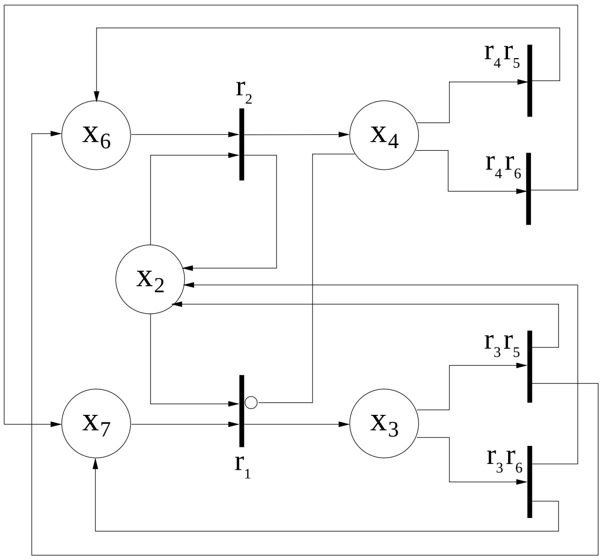

Fig. 2. Readers-writers protocol with fused transitions

with more than four places, in general do not have a a reachability set expressible in linear arithmetic [19]. For the special class of BPP nets, the situation is however different. Recall that a Petri net is a BPP-net if every transition has at most [1] one input place and removes exactly one token from that one place. BPP stands for Basic Parallel Process: this is a class of CSS process defined in [6]; the reachability problem for BPP-nets is NP-complete [11]. When one encodes the reachability problem for BPP-nets, using the method of section 3, one obtains a program such that, for any clause $r_i \in \Sigma$, all the coefficients of the head are nonnegative except (maybe) one, which is equal to -1. For all clause r_i, if such a negative coefficient, say $k_{i,h}$, exists, then the constraint of r_i is atomic and equal to $x_h > 0$. For BPP-nets,one thus shows (see [14]) that the decomposition process is guaranteed to terminate successfully, and yield a flat language. The decomposition method thus shows that the *binary* relation $\overline{x} \xrightarrow{\Sigma^*} \overline{x}'$ is an existentially quantified Presburger formula having $\overline{x}$ and $\overline{x}'$ as free variables (that is, $\{\langle \overline{x}, \overline{x}' \rangle \mid \overline{x} \xrightarrow{\Sigma^*} \overline{x}'\}$ is a semilinear set (see [16]) This gives a new proof of the fact that the reachability set for BPP-nets is a semilinear set [11].

[1] The original definition states that every transition has *exactly* one input place, but it is convenient here to relax it somewhat.

5 Compilation into Presburger Arithmetic

Hans Olsén has written an experimental implementation of the system in SICStus Prolog containing seven decomposition rules, two of which are cyclic (*cf.* [24,13]). Besides the decomposition module, it contains a theorem prover for Presburger arithmetic based on the decision procedure of [4]. The system outputs a regular expression defining a flat language (if the decomposition is successful) and constructs a graph representation of the corresponding Presburger formula (see [4]). For the readers-writers protocol of figure 2, our system finds a flat language $L \subseteq \Sigma^*$ such that $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Leftrightarrow\; \overline{x} \xrightarrow{L} \overline{x}'$, which is:

$$\begin{aligned} L = {} & r_5^* r_6^* r_2^* r_1^* r_2^* r_4^* (r_2 r_4)^* r_5^* (r_2 r_4)^* r_5^* r_6^* r_2^* r_1^* r_2^* r_3^* (r_1 r_3)^* r_2^* r_4^* (r_1 r_3)^* (r_2 r_4)^* \\ & (r_1 r_3)^* r_2^* r_5^* r_6^* (r_1 r_3 r_5)^* r_2^* (r_4 r_6)^* (r_4 r_5)^* (r_1 r_3 r_5)^* (r_2 r_4 r_6)^* (r_1 r_3 r_5)^* \\ & (r_1 r_3)^* r_2^* r_4^* (r_1 r_3)^* (r_2 r_4)^* (r_1 r_3)^* r_5^* r_6^* r_2^* r_1^* r_2^* r_4^* (r_2 r_4)^* r_5^* (r_2 r_4)^* r_5^* r_6^* \\ & r_2^* r_1^* r_2^* \end{aligned}$$

The expression consists of 51 factors and was computed in 0.43 seconds on a SPARC-10 machine. The decomposition was achieved by using 7 applications of the cyclic post-fusion rule presented above, 10 applications of the stratification rule and 1 application of monotonic guard (see [13]).
Let us denote the above language L as $w_1^* w_2^* \cdots w_{51}^*$. When computing the least fixed-point of a program, we are interested in the set $post^*(B)$: $\{\overline{x}' \mid \exists \overline{x} : B(\overline{x}) \wedge \overline{x} \xrightarrow{w_1^* w_2^* \cdots w_{51}^*} \overline{x}'\}$. We are thus led to construct a sequence $\{\xi_i(\overline{x})\}_{i=0,\ldots,51}$ of relations defined by:

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

We have: $\overline{x} \in post^*(B) \quad \Leftrightarrow \quad \xi_{51}(\overline{x})$. The decision procedure for Presburger arithmetic is invoked to construct the sequence of ξ_is. Actually, a dynamic test of "invariance" is added during the compilation process into arithmetic in order to check wether the fixpoint has already been generated. That is, for each i $(0 \leq i \leq 50)$, one checks wether

$\forall r \in \{r_1, ..., r_6\} : \ \xi_i(\overline{x}) \wedge \vartheta_r(\overline{x}) \ \Rightarrow \ \xi_i(\overline{x}r)$.

If this is true, there is no need to continue, and this may significantly reduce the size of the final expression. This corresponds to the test of *subsumption*, as used in constraint databases or bottom-up Constraint Logic Programming (see, e.g., [22]). In practice, this test is done before a set of clauses is decomposed. For example, if the monotonic decomposition rule of proposition 4 can be applied to Σ with some rule r, one has to consider reachability by $(\Sigma - \{r\})^* r^* (\Sigma - \{r\})^*$. Before decomposing the leftmost subexpression, one checks wether all clauses in $\Sigma - \{r\}$ lets the set ξ invariant. If so, one goes on to $r^*(\Sigma - \{r\})^*$. If r does not leave ξ invariant, one computes a new set ξ' and before decomposing the final set $\Sigma - \{r\}$ one checks if all clauses in $\Sigma - \{r\}$ leaves ξ' invariant etc. Let us illustrate such a fixed-point computation on program P' of example 2 from the language $L = r_5^* r_6^* r_2^* r_1^* \ldots$, generated by our program. We get the sequence (making arithmetic simplifications at each step):

$$\begin{array}{ll}
\xi_0(\overline{x}) \Leftrightarrow B(\overline{x}) & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \ x_2 = 1 \ \wedge \ x_3 = 0 \wedge \\
& \quad x_4 = 0 \ \wedge \ x_5 = q \ \wedge \ q \geq 0 \wedge \\
& \quad x_6 = 0 \ \wedge \ x_7 = 0 \\
\xi_1(\overline{x}) \Leftrightarrow \exists \overline{x}'' : \ \xi_0(\overline{x}'') \wedge \overline{x}'' \xrightarrow{r_5^*} \overline{x} & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \ x_2 = 1 \ \wedge \ x_3 = 0 \wedge \\
& \quad x_4 = 0 \ \wedge \ x_5 \geq 0 \ \wedge \ x_6 \geq 0 \wedge \\
& \quad x_7 = 0 \ \wedge \ x_5 + x_6 = q \\
\xi_2(\overline{x}) \Leftrightarrow \exists \overline{x}'' : \ \xi_1(\overline{x}'') \wedge \overline{x}'' \xrightarrow{r_6^*} \overline{x} & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \ x_2 = 1 \ \wedge \ x_3 = 0 \wedge \\
& \quad x_4 = 0 \ \wedge \ x_5 \geq 0 \ \wedge \ x_6 \geq 0 \wedge \\
& \quad x_7 \geq 0 \ \wedge \ x_5 + x_6 + x_7 = q \\
\xi_3(\overline{x}) \Leftrightarrow \exists \overline{x}'' : \ \xi_2(\overline{x}'') \wedge \overline{x}'' \xrightarrow{r_2^*} \overline{x} & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \ x_2 = 1 \ \wedge \ x_3 = 0 \wedge \\
& \quad x_4 \geq 0 \ \wedge \ x_5 \geq 0 \ \wedge \ x_6 \geq 0 \wedge \\
& \quad x_7 \geq 0 \ \wedge \ x_4 + x_5 + x_6 + x_7 = q \\
\xi_4(\overline{x}) \Leftrightarrow \exists \overline{x}'' : \ \xi_3(\overline{x}'') \wedge \overline{x}'' \xrightarrow{r_1^*} \overline{x} & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \\
& \quad ((x_2 = 1 \ \wedge \ x_3 = 0 \ \wedge \ x_4 \geq 0) \ \vee \\
& \quad\ (x_2 = 0 \ \wedge \ x_3 = 1 \ \wedge \ x_4 = 0)) \ \wedge \\
& \quad x_5 \geq 0 \ \wedge \ x_6 \geq 0 \ \wedge \ x_7 \geq 0 \wedge \\
& \quad x_3 + x_4 + x_5 + x_6 + x_7 = q
\end{array}$$

One may easily check that $\forall r_i \in \{r_1, \ldots, r_6\} : \ \vartheta_{r_i}(\overline{x}) \wedge \xi_4(\overline{x}) \ \Rightarrow \ \xi_4(\overline{x} r_i)$. This means that the fixed-point has been reached. The fixed-point of the program P' is thus:

$$post^*(B) = \{\overline{x}' \mid \exists \overline{x} : \ B(\overline{x}) \wedge \overline{x} \xrightarrow{r_5^* r_6^* r_2^* r_1^*} \overline{x}'\}$$

and out of the 51 factors in the language L, only the first 4 are needed. We have:

$$\begin{array}{ll}
\overline{x} \in post^*(B) \ \Leftrightarrow \ \xi_4(\overline{x}) & \Leftrightarrow x_1 = 1 - x_4 \ \wedge \\
& \quad ((x_2 = 1 \ \wedge \ x_3 = 0 \ \wedge \ x_4 \geq 0) \ \vee \\
& \quad\ (x_2 = 0 \ \wedge \ x_3 = 1 \ \wedge \ x_4 = 0)) \ \wedge \\
& \quad x_5 \geq 0 \ \wedge \ x_6 \geq 0 \ \wedge \ x_7 \geq 0 \wedge \\
& \quad x_3 + x_4 + x_5 + x_6 + x_7 = q
\end{array}$$

It is easy to see that the mutual exclusion property holds ($\overline{x} \in post^*(B) \Rightarrow x_3 = 0 \ \vee \ x_4 = 0$). Olsén's program constructs the arithmetical form of the least fixed-point in 1.1 second, and proves the property in 0.5 second. Details on the integration of Boudet-Comon decision procedure for linear arithmetic (coupled with bottom-up evaluation with subsumption) into the basic decomposition process can be found in [13,24].

6 Real arithmetic

The complexity of Presburger arithmetic is extremely high, a major obstacle for dealing with more than small-size examples. Treating integers as reals is a way to alleviate this complexity problem. It is well known that real arithmetic is more tractable than integer arithmetic: for instance, checking the existence of a solution over $\mathbb{R}$ for a system of linear inequations is polynomial while it is NP-hard over $\mathbb{N}$ [25]. In general resolution over $\mathbb{R}$ leads to an *upper approximation* of the set of integer solutions. However Shostack [26] noticed that, for a particular class of formulas (made of constraints of the form $x+c \leq y$, $x \leq c$ or $c \leq x$), elimination of variables over $\mathbb{R}$ using Fourier-Motzkin procedure is *exact*, i.e. produces the same result as elimination of variables over $\mathbb{N}$. For this class, it is safe to focus on "polyhedric" constraints (i.e. sets of linear inequalities), and discard the cumbersome divisibility constraints that are *a priori* needed when dealing with integers. Following Shostack, it can be shown that reachability analysis of Petri nets can be done by treating integer variables as real-valued variables, and eliminating them with Fourier-Motzkin, as far as one replaces all the constraints of the form $x > a$ with $x \geq a+1$. Another condition is that, when eliminating the existential variable μ in the expression $\overline{x} \xrightarrow{w^*} \overline{x}'$ (see proposition 1), all the coefficients of μ in lower bounds on μ are 1 (or all the coefficients of μ in upper bounds on μ are 1). Under these conditions, given a Petri net and an initial marking, iterative computation of the successor markings with Fourier-Motzkin yields the exact reachability set *post**, when it terminates. The same is true for backward computation of predecessors *pre**. Otherwise, one obtains an upper-approximation of theses sets.

In the following examples we use a two-step method. First we construct an extended set Σ' obtained from Σ by adding fused transitions. This is done using Olsén's module of decomposition (see [24]). The decomposition is performed up to a certain level k, i.e. until k distinct fused transitions have been generated. The choice of the appropriate value k is the result of a compromise: the bigger k is, the better the convergence of *post** or *pre** is likely to be, but the space search may explode for too large values of k. Given Σ', one computes the set *post** (resp. *pre**) of iterated successors (resp. predecessors) via Σ' in a separate step of bottom-up evaluation. [2] In the following examples, the tool HYTECH [18] is used for computing *pre**. All the variables are implicitly nonnegative. See [1] for details.

Example 7. The Petri net in figure 3 corresponds to the "swimming-pool" example from M. Latteux (see [5,7]). All weights are equal to 1. Transition r_1 (incoming guest) is enabled if $x_6 \geq 1$, when there is at least one free cabine. When r_1 is fired, one token is removed from counter x_6 and added into counter x_1, meaning the beginning of undressing. The initial marking involves two parameters p_1 and p_2 (assumed to be greater than or equal to 1), which represent the number of available cabins and baskets respectively. We have $B:\ x_1 = x_2 = x_3 = x_4 = x_5 = 0\ \wedge\ x_6 = p_1 \wedge x_7 = p_2 \wedge p_1 \geq 1 \wedge p_2 \geq 1$.

[2] As seen above, the decomposition process also infers an ordering of application among the set of ordinary and fused transitions, but this information is not exploited here.

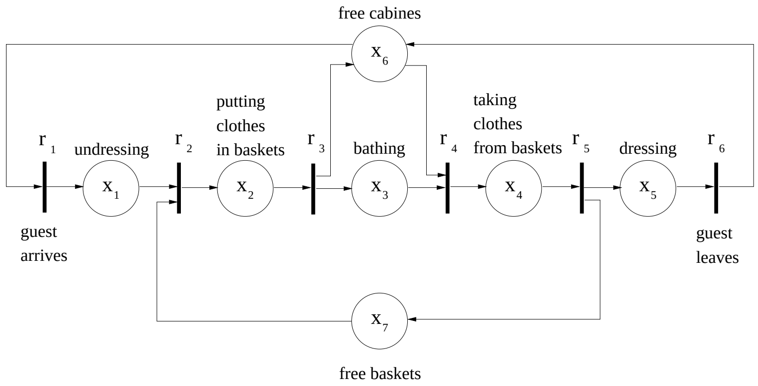

Fig. 3. A Petri net for the swimming pool example

Consider the set of transitions $\Sigma = \{r_1, r_2, \cdots, r_6\}$ for the swimming pool Petri net. The Petri net derived by decomposition has a set Σ' with the following 6 fused transitions $\{r_2r_3,\ r_1r_2r_3,\ r_6r_4,\ r_4r_5,\ r_5r_6r_4,\ r_1r_2r_3r_4r_5\}$ added to Σ. The deadlock formula, characterizing the set of markings from which no transition can be fired, is of the form $F_{dead_1} \vee F_{dead_2}$ with:

$F_{dead_1} : x_2 = x_4 = x_5 = x_6 = x_7 = 0, \quad F_{dead_2} : x_1 = x_2 = x_4 = x_5 = x_6 = 0.$

The bottom-up evaluation of pre^* via Σ' starting from F_{dead_1} terminates after 22 steps (and takes 50 seconds). Besides, one can check that all the nonnull coefficients of μ appearing in lower bounds on μ on the intermediate formulas, are equal to 1. Therefore the result of pre^* is exact. It corresponds to the disjunction of the following 10 formulas:

$\{x_2 = x_4 = x_5 = x_6 = x_7 = 0, \quad x_2 = x_4 = x_5 = x_7 = 0 \wedge x_6 \geq 1,$
$x_2 = x_4 = x_7 = 0 \wedge x_5 \geq 1, \quad x_2 \geq 1, \quad x_1 \geq 1 \wedge x_7 \geq 1, \quad x_6 \geq 1 \wedge x_7 \geq 1,$
$x_4 \geq 1, \quad x_3 \geq 1 \wedge x_6 \geq 1, \quad x_5 \geq 1 \wedge x_7 \geq 1, \quad x_3 \geq 1 \wedge x_5 \geq 1\}$

One of these formulas, $x_6 \geq 1 \wedge x_7 \geq 1$, has B as an instance. It follows that, for any values of parameters p_1 and p_2 (greater than or equal to 1), there exists a deadlock. It is often interesting to keep track of a path linking an initial marking to a deadlock, but this requires to store additional information during the construction of pre^* at a cost generally prohibitive. In this simple example however, it is possible to retrieve such a path: for all values of the parameters p_1 and p_2 of the initial marking, there is a path of the form $(r_1r_2r_3)^*r_1^*$ that goes from a point of B to a point of F_{dead_1}.

Example 8. This is a partial modelling of communication protocol PNCSA (Standard Protocol for Connection to the Authorization System [15]), borrowed from [12]. The Petri net has 38 transitions and 31 counters. The initial marking has two components e and f equal to 1, and all the other ones null. The deadlock formula is of the form: $F_{dead_1} \vee \cdots \vee F_{dead_4}$. The extended Petri net Σ' is obtained by decomposing Σ up to level 20, which yields fused transitions of the

form: $\sigma_1 \equiv r_1 r_2 r_3$, $\sigma_2 \equiv r_{21} r_{34} r_{37}$,
$\sigma_3 \equiv r_1 r_{14} r_{17} r_{21} r_{34} r_{37}$, $\sigma_4 \equiv r_{12} r_{13} r_{17} r_{21} r_{22} r_{31} r_{37}, \quad \cdots$.
The bottom-up evaluation of pre^* via Σ' starting from F_{dead_1} terminates after 53 steps, and takes 15 minutes. It contains 376 constraints. One can also check that all the nonull coefficients of μ appearing in lower bounds on μ of the intermediate formulas, are equal to 1. Therefore the result of pre^* is exact. Here again $pre^*(F_{dead_1})$ contains an instance of B. This proves that the system has a deadlock. It is here also possible to exhibit a path linking an initial marking to a deadlock marking (see Appendix). Note that the existence of this deadlock was unknown so far.

As explained above, these two examples were run with HYTECH, after a preprocessing with Olsén's decomposition tool (*DECO* available at `http://www.ida.liu.se/labs/logpro/sw.shtml`). See [1] for further examples. Recently, Guy Narboni obtained interesting results, following a similar approach, but using a prototype written in PROLOG III instead of HYTECH [23]. Other tools like *DMC* ("Deductive Model Checking", see [9]) make also use of Constraint Logic Programming and Real Arithmetic for proving properties of concurrent systems, but in the absence of additional fused transitions, they fail to treat sophisticated Petri net such as the above examples. This emphasizes the importance of decomposing the set of transitions into flat languages throuh operations like fusion.

References

1. B. Bérard and L. Fribourg. "Reachability analysis of (timed) Petri nets using real arithmetic". *Proc. CONCUR'99*, LNCS 1664, Springer-Verlag, 1999, pp. 178-193.
2. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S. and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, Chichester,1995.
3. Berthelot, G., "Transformations and Decompositions of Nets". *Advances in Petri Nets*, LNCS 254, Springer-Verlag, 1986, pp. 359-376.
4. Boudet, A. and H. Comon. "Diophantine Equations, Presburger Arithmetic and Finite Automata". *Proc. 21st Intl. Colloquium on Trees in Algebra and Programming*, LNCS 1059, Springer-Verlag, 1996, pp. 30-43.
5. Brams, G.W., *Réseaux de Petri: Théorie et Pratique*, Masson, Paris, 1983.
6. Christensen, S., *Decidability and Decomposition in Process Algebras.* Ph.D. Thesis, University of Edinburgh, CST-105-93, 1993.
7. R. David and H. Alla. *Du Grafcet aux Réseaux de Petri*, Hermès, Paris, 1989.
8. Cousot, P. and N. Halbwachs, "Automatic Discovery of Linear Restraints among Variables of a Program". *Conference Record 5th ACM Symp. on Principles of Programming Languages*, Tucson, 1978, pp. 84-96.
9. G. Delzanno and A. Podelski. "Model-Checking in CLP". *Proc. TACAS'99*, LNCS 1579, Springer-Verlag, 1999, pp. 223-239.
10. Esparza, J. and M. Nielsen, "Decidability Issues for Petri Nets". Bulletin of the EATCS, Number 52, Feb. 1994.
11. Esparza, J., "Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes". *Proc. of Fundamentals of Computer Theory '95*, LNCS 965, 1995.

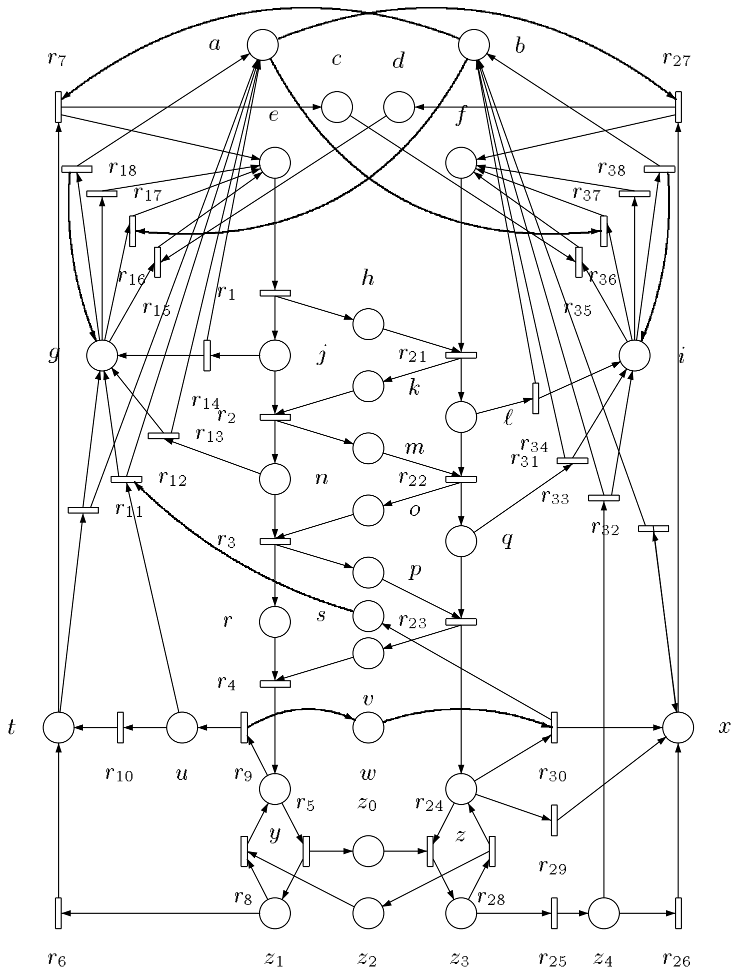

Fig. 4. Communication protocol PNCSA

12. A. Finkel. "The Minimal Coverability Graph for Petri Nets". *Advances in Petri Nets.* LNCS 674, Springer-Verlag, 1993, pp. 211–243.
13. Fribourg, L. and H. Olsén, "Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic". *Proc. CONCUR'97*, LNCS 1243, Springer-Verlag, 1997, pp. 213–227.
14. Fribourg, L. and H. Olsén, "A Decompositional Approach for Computing Least Fixed-Points of datalog Programs with $\mathbb{Z}$-counters". *Constraints 2*, Kluwer, 1997, pp. 305–335.
15. GIE CB. "Protocole normalisé de connexion au système d'autorisation: spécifications PNCSA Version 2", document C *Communication OSI*, 1988.
16. Ginsburg, S. and E.H. Spanier, "Semigroups, Presburger formulas and languages". *Pacific Journal of Mathematics 16*, 1966, pp. 285-296.
17. Halbwachs, N., "Delay Analysis in Synchronous Programs". *Proc. Computer Aided Verification*, LNCS 697, Springer-Verlag, 1993, pp. 333-346.
18. T. Henzinger, P.-H. Ho and H. Wong-Toi. "A User Guide to HYTECH". *Proc. TACAS'95*, LNCS 1019, Springer-Verlag, 1995, pp. 41–71.
19. Hopcroft, J. and J.-J. Pansiot, "On the Reachability Problem for 5-dimensional Vector Addition Systems". *Theoretical Computer Science*, vol. 8, 1979, pp. 135-159.
20. Jaffar, J. and J.L. Lassez, "Constraint Logic Programming". *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 111-119.
21. Kanellakis, P., Kuper, G., 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).
22. Maher, M.J., "A Logic Programming View of CLP". *Proc. 10th Intl. Conf. on Logic Programming*, Budapest, 1993, pp. 737-753.
23. G. Narboni. "Verification of Petri net properties in CLP". *Proc. Intl. Workshop on Verification and Computation Logic*, Tech. Report DSSE-TR-2000-6, 2000.
24. H. Olsén. *Automatic Verification of Petri Nets in a CLP framework.* Ph.D Thesis, University of Linköping, Sweden, 1997. (available at `http://www.ida.liu.se/labs/logpro/`)
25. A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, 1986.
26. R. Shostack. "Deciding linear inequalities by computing loop residues". *J. ACM* 28:4, 1981, pp. 769–779.
27. Yen, H.-C., "On the Regularity of Petri Net Languages". *Information and Computation 124*, 1996, pp. 168-181.

Appendix: Exhibition of a deadlock in PNCSA

In the initial formula B, the counters e, f are 1 and all the other counters 0. The deadlock (sub)formula F_{dead_1} is:
$e = g = h = i = j = \ell = n = q = z = z_1 = t = u = v = x = y = z_3 = z_4 = 0$.

We have: $\overline{x} \xrightarrow{\sigma_1^\mu} \overline{x}'$ iff $\quad b' = b + \mu \wedge h' = h - \mu \wedge k' = k + \mu \wedge f \geq 1 \wedge h \geq \mu.$

We have: $\overline{x} \xrightarrow{\sigma_2^\mu} \overline{x}'$ iff

$$e' = e - \mu \wedge h' = h + \mu \wedge k' = k - \mu \wedge m' = m + \mu \wedge o' = o - \mu \wedge p' = p + \mu \wedge r' = r + \mu \wedge e \geq \mu \wedge k \geq \mu \wedge o \geq \mu.$$

We have: $\overline{x} \xrightarrow{\sigma_3^\mu} \overline{x}'$ iff $\quad a' = a + \mu \wedge b' = b + \mu \wedge k' = k + \mu \wedge e \geq 1 \wedge f \geq 1.$

We have: $\overline{x} \xrightarrow{\sigma_4^\mu} \overline{x}'$ iff

$$a' = a + \mu \wedge b' = b + \mu \wedge o' = o + \mu \wedge e \geq 1 \wedge f \geq 1 \wedge k \geq 1.$$

We have the following predecessor formulas.

$pre_{\sigma_1^\mu}(F_{dead_1})$ is:

$$\varphi' : \ z_4 = z_3 = u = z_1 = e = x = g = t = i = j = v = \ell = z = n = y = q = 0 \wedge h = \mu \wedge b + h \geq 0 \wedge h + k \geq 0.$$

Elimination of μ in $(\exists \mu \geq 1\ \varphi')$ gives:

$$\varphi_1 : \ e = g = i = j = \ell = n = q = t = u = v = x = y = z = z_1 = z_3 = z_4 = 0 \wedge f \geq 1 \wedge h \geq 1$$

$pre_{\sigma_2^\mu}(\varphi_1)$ is:

$$\varphi_1' : \ z_3 = v = z_1 = x = u = t = g = i = j = y = \ell = z = n = q = z_4 = 0 \wedge f \geq 1 \wedge e = \mu \wedge k \geq e \wedge o \geq e \wedge e + r \geq 0 \wedge e + h \geq 1 \wedge e + p \geq 0 \wedge e + m \geq 0.$$

Elimination of μ in $(\exists \mu \geq 1\ \varphi_1')$ gives:

$$\varphi_2 : \ g = i = j = \ell = n = q = t = u = v = x = y = z = z_1 = z_3 = z_4 = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge k \geq e \wedge o \geq e.$$

$pre_{\sigma_3^\mu}(\varphi_2)$ is:

$$\varphi_2' : \ v = z_1 = t = z = z_3 = q = g = z_4 = i = j = y = \ell = x = n = u = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge o \geq e \wedge k + \mu \geq e \wedge a + \mu \geq 0 \wedge b + \mu \geq 0.$$

Elimination of μ in $(\exists \mu \geq 1\ \varphi_2')$ gives:

$$\varphi_3 : \ g = i = j = \ell = n = q = t = u = v = x = y = z = z_1 = z_3 = z_4 = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge o \geq e.$$

$pre_{\sigma_4^\mu}(\varphi_3)$ is:

$$\varphi_3' : \ u = v = z_4 = z_3 = g = t = i = j = y = \ell = z_1 = n = x = q = z = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge a + \mu \geq 0 \wedge b + \mu \geq 0 \wedge o + \mu \geq e.$$

Elimination of μ in $(\exists \mu \geq 1\ \varphi_3')$ gives:

$$\varphi_4 : \ g = i = j = \ell = n = q = t = u = v = x = y = z = z_1 = z_3 = z_4 = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge k \geq 1.$$

$pre_{\sigma_3^\mu}(\varphi_4)$ is:

$$\varphi_4' : \ u = v = z_4 = z_3 = g = t = i = j = y = \ell = z_1 = n = x = q = z = 0 \wedge e \geq 1 \wedge f \geq 1 \wedge a + \mu \geq 0 \wedge b + \mu \geq 0 \wedge k + \mu.$$

Elimination of μ in $(\exists \mu \geq 1\ \varphi_4')$ gives:

$$\varphi_5 : \ g = i = j = \ell = n = q = t = u = v = x = y = z = z_1 = z_3 = z_4 = 0 \wedge e \geq 1 \wedge f \geq 1.$$

Clearly φ_5 has B as an instance. This shows that there is a path from B to an instance of F_{dead_1} of the form $(r_1 r_{14} r_{17} r_{21} r_{34} r_{37})^{\mu_1}$ $(r_1 r_2 r_{13}\ r_{17} r_{21} r_{22} r_{31} r_{37})^{\mu_2}$ $(r_1 r_{14} r_{17} r_{21} r_{34} r_{37})^{\mu_3}$ $(r_1 r_2 r_3)^{\mu_4}$ $(r_{21} r_{34} r_{37})^{\mu_5}$ for some $\mu_1, \cdots, \mu_5 \geq 1$. For $\mu_1 = \cdots = \mu_5 = 1$, the path leads to the instance of F_{dead_1} with $a = 3, b = 4, k = 2, f = m = p = r = 1$ and all the other counters null.