, , 1?? ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Decompositional Approach for Computing Least Fixed-Points of Datalog Programs with $\mathcal{Z}$-Counters

LAURENT FRIBOURG fribourg@lsv.ens-cachan.fr
Ecole Normale Supérieure Cachan & CNRS, 61 av. Président Wilson, 94235 Cachan - France

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

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 a set of rules that transform general computation paths into "canonical" ones. We use the method for treating the problem of reachability in the field of Petri nets, thus relating some unconnected results and extending them in several directions.

Keywords: decomposition, linear arithmetic, least fixed-point, Petri nets, reachability set

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 (Cousot, 78). The subject has known a renewal of interest with the development of logic programming and deductive databases with arithmetical constraints. Several new applications were then possible in these frameworks: proof of termination of logic programs (Van Gelder, 90)(Plumer, 90)(Verschaetse, 91), compilation of recursive queries in temporal databases (Baudinet, 91)(Kabanza, 90), verification of safety properties of concurrent systems (Halbwachs, 93). However almost all these works are interested in finding not the *least* fixpoint but rather an approximation of it using some techniques of Abstract Interpretation (convex hull, widening, ...). A notable exception is the work of (Revesz, 90) and of (Chomicki, 88) whose procedures allow to compute least fixpoints, but for very restrictive classes of programs, viz. programs with no or at most one incremental argument. In this paper we are interested in finding the *least* fixed points for Datalog programs 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. The main difference is that the variables of our programs take their values on $\mathcal{Z}$ instead of $\mathcal{N}$ 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 (Berthelot, 86), still apply to our programs with $\mathcal{Z}$-counters. The fact that we manipulate variables taking their values on $\mathcal{Z}$ rather than on $\mathcal{N}$ will allow us to encode in a simple way the important extension of Petri nets with *inhibitors*. As an example, we will see how to prove the mutual exclusion property of a Petri net modelling a system of readers and writers where the number of processes is parametric. We also show how our method allows us to treat the reachability problem for a special class of Petri nets, called BPP-nets, thus generalizing a result of (Esparza, 95).

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 compares our approach with relevant work. Section 6 shows that our method allows us to solve the reachability problem for the special class of Basic Parallel Process nets. Section 7 gives a further generalized form of our basic decomposition rule. Section 8 briefly discusses the compilation into an arithmetic formula and our implementation. We conclude in section 9.

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, $k_{i,j}, a_{i,l} \in \mathcal{Z}$ 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 $\mathcal{Z}$-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 $\mathcal{Z}$-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:

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

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:

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

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 (borrowed from an example given later on):

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

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

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

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. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow \ \exists\, n \geq 0 : \ \overline{x}' = \overline{x} + n \cdot \overline{k}_w \ \wedge \ \forall\, 0 \leq n' < n : \ \vartheta_w(\overline{x} + n' \cdot \overline{k}_w)$

In part 3 of the proposition, when $n = 0$, we have $\overline{x} = \overline{x}'$, and the expression $\forall\ 0 \leq n' < n :\ \vartheta_w(\overline{x} + n' \cdot \overline{k}_w)$ is vacuously true. Actually, the universally quantified variable n' 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, $\vartheta_w(\overline{x} + n' \cdot \overline{k}_w)$ is $\overline{x} + n' \cdot \overline{k}_w \geq \overline{a}_w$. It is easy to see that, for $n > 0$, the universally quantified expression $\forall\ 0 \leq n' < n :\ \overline{x} + n' \cdot \overline{k}_w > \overline{a}_w$ is equivalent to $\overline{x} + (n-1) \cdot \overline{k}_w^{-} > \overline{a}_w$ 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 $\forall\ 0 \leq n' < n :\ \overline{x} + n' \cdot \overline{k}_w > \overline{a}_w$ is

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

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 (Jaffar, 87)(Kanellakis, 90)), which is also the least $\mathcal{Z}$-model of the program, 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 $\overline{y} \in \text{lfp}$ as an arithmetic formula having $\overline{y}$ as a free variable. For solving this problem, it suffices actually to characterize the 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, e.g., (Yen, 96)). The rest of the paper is mainly devoted to describe the decomposition rules that we use and their applications for solving the reachability problem with Petri nets and some of their extensions.

3. The Reachability Problem for Petri Nets

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

There is a close connection between the class of programs with $\mathcal{Z}$-counters and Petri nets, and more precisely, between the computation of the least fixed-point of programs with $\mathcal{Z}$-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 (Esparza, 94).) 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 $\mathcal{N}$ 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 $\mathcal{Z}$-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 $\mathcal{Z}$-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 $\mathcal{N}$ of non-negative integers while the domain for programs with $\mathcal{Z}$-counters is $\mathcal{Z}$. From a theoretical point of view, programs with $\mathcal{Z}$-counters have the power of Turing machines while (programs coding for the reachability problem of) Petri nets have not. We will come back to this issue in the forthcoming subsection.

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 $\mathcal{Z}$-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. When the input place is known to be bounded (i.e., the place can never contain more than a fixed number of tokens during the evolution of the Petri net configuration), it is well-known that one can simulate such a 0-test using conventional Petri nets. For example, if the bound of the inhibitor place is known to be 1, it is easy to add a "complementary place" to the net whose value is 0 (resp. 1) when the inhibitor place is 1 (resp. 0). Instead of testing the inhibitor place to 0, it is equivalent to test if the complementary place contains (at least) one token. Such a simulation is not possible when the place is unbounded. Actually Petri nets with inhibitor places can simulate Turing machines, so there is no hope to simulate such an extension while keeping inside the class of Petri nets.

On the other hand, 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'^0_j 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 $\mathcal{Z}$-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

Recall that the least fixed-point of the encoding program (i.e., the reachability set of the corresponding Petri net) can be expressed as follows:

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

Here $B(\overline{x})$ is $\overline{x} = \overline{a}^0$ where $\overline{a}^0$ denotes the initial marking of the Petri net, that is *a priori* a tuple of nonnegative constants. Our aim is to characterize the relation $\overline{y} \in \text{lfp}$ as an arithmetical formula having $\overline{y}$ as a free variable. It is however often interesting to reason more 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 $\mathcal{Z}$-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 $\overline{y} \in \text{lfp}$ as an arithmetical formula having $\overline{y}$ and $\overline{q}$ as free variables.

3.4. Example

We illustrate the encoding of Petri-nets with inhibitors and parametric initial markings by an example.

EXAMPLE: We consider here a Petri net implementing a simple readers-writers protocol. (This is inspired from (Ajmone, 95), 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:

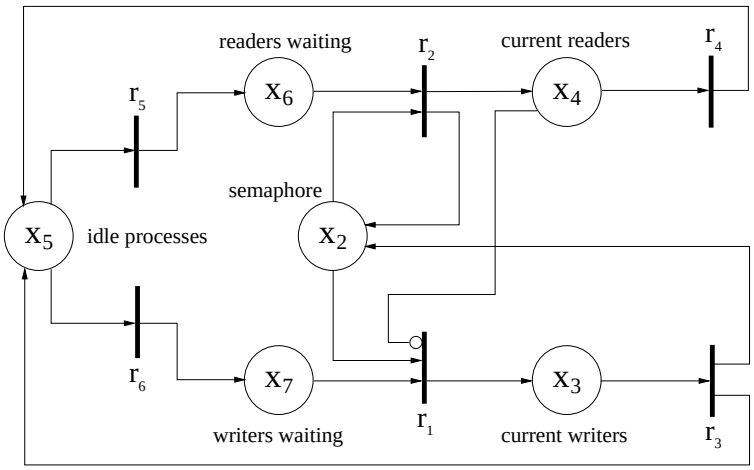


Figure 1

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

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 that, for this protocol, there is always at most one current writer (i.e. $x_3 = 0 \vee x_3 = 1$), and that reading and writing can never occur at the same time (i.e.: $x_3 = 0 \vee x_4 = 0$). Formally, we must prove:

$$p'(x_1,x_2,x_3,x_4,x_5,x_6,x_7) \ \Rightarrow\ (x_3 = 0 \vee x_3 = 1)$$

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

The classical methods of verification of Petri nets by invariants (see, e.g., (Brams, 83)(Peterson, 81)) are able to prove the first implication: by analysing the transitions without taking into account the guards, they generate a set of linear combinations $\Sigma_{i=1}^{7}\lambda_i x_i$ of $x_1, ..., x_7$, which are left invariant by any transition r_j ($1 \leq j \leq 6$)[1]. Among the generated invariants, there is the formula $x_2 + x_3 = 1$. Since the variables x_2 and x_3 take only positive or null values, it follows immediately that x_3 must be 0 or 1. The second property of "mutual exclusion" ($x_3 = 0 \vee x_4 = 0$) is more difficult to establish. (See however (Meltzer, 95) for a recent method extending the classical methods with invariants for dealing with such mutual exclusion properties.) We will see in this paper how our method of construction of least fixed-points allows us to solve this problem (see section 8).

□

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: The program P' of example 3.4 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 "simpler" form. In this paper we present in detail only two transformation rules. (The definition of some other rules is given besides in appendix A.) 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 "simpler" program. Programs Σ_i are "simpler" 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.)

Formally, we define the *dimension* of a program with $\mathcal{Z}$-counters as a couple (m, n) where n is the number of clauses of the program, and m is the number of *non invariant* variables of the program (i.e. the number of non null columns in the corresponding incrementation matrix). We also define an order on these dimensions as follows: The dimension (m_1, n_1) is lower than (m_2, n_2) iff $m_1 < m_2$, or $m_1 = m_2$ and $n_1 < n_2$. Each transformation rule thus decomposes the original language Σ^* into either finite languages (for which the reachability problem is solvable in the existential fragment of Presburger arithmetic, see section 2) or into languages associated with programs of lower dimension. There are two kinds of "elementary" programs with a *basic* dimension. The first kind consists in programs of dimension $(1, n)$, i.e. programs made of n clauses, $r_1, \ldots, r_n$ with all but one column being null. As will be seen later on (see section 4.1.1, remark 3), the reachability problem

for such programs can be easily solved and expressed in the existential fragment of Presburger arithmetic. The second kind of elementary programs are programs of dimension $(m, 1)$, i.e., programs made of a single clause, say r_1. In this case the expression $\overline{x} \xrightarrow{\Sigma^*} \overline{x}'$ reduces to $\overline{x} \xrightarrow{r_1^*} \overline{x}'$, which can be also expressed in the existential fragment of Presburger arithmetic (see section 2). Therefore the decomposition process must eventually terminate either successfully, thus leading to a characterization of the reachability relation in Presburger arithmetic, or it terminates with failure because no decomposition rule can be applied.

In keeping with a former approach (Fribourg, 94), we will consider two types of decomposition rules: *monotonic* and *cyclic* rules. The monotonic decompositions are based on the fact that some clauses of a program may be applied all at once at some point during a computation, while the cyclic decompositions 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:

$$\begin{array}{ccccc} & \vdots & & \vartheta_{r_{\dots}} & : r_{\dots} \\ + & \dots & + & \vartheta_{r_l} & : r_l \\ & \vdots & & \vartheta_{r_{\dots}} & : r_{\dots} \\ x_1 & & x_m & & \end{array}$$

This means that, in the program, we have $\forall j : k_{l,j} \geq 0$. 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 $\vartheta_{r_l}(\overline{x})$ holds, one can always reorder w so that all the clauses r_l are applied first. Formally we have:

PROPOSITION 2 *Let $r_l \in \Sigma$ be a clause such that $\forall j : k_{l,j} \geq 0$. Then:*

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

Proof: Since $\overline{k}_{r_l}$ is a vector made of nonnegative coefficients $k_{l,j}$, we have: $\vartheta_{r_j}(\overline{x}) \;\Rightarrow\; \vartheta_{r_j}(\overline{x} + \overline{k}_{r_l})$, i.e. $\vartheta_{r_j}(\overline{x}) \;\Rightarrow\; \vartheta_{r_j}(\overline{x} r_l)$, for all $r_j \in \Sigma$. The constraint ϑ_{r_j} is thus invariant under the application of r_l. Therefore, if $\overline{x}'$ is reachable from $\overline{x}$ by some path $w = w_1 r_l w_2$, and $\vartheta_{r_l}(\overline{x})$ holds, then also the path $w' = r_l w_1 w_2$ is applicable, so all the applications of r_l can be pushed to the beginning, and thus $\overline{x}'$ must be reachable from $\overline{x}$ by some path $w'' = r_l \cdots r_l w_3$ where $w_3 \in (\Sigma - \{r_l\})^*$.

Clearly, if $\overline{x}'$ is reachable from $\overline{x}$ by any path $w \in \Sigma^*$ containing r_l, then r_l must occur somewhere for the first time. At that point ϑ_{r_l} must hold, so, by the above, $\overline{x}'$ is reachable by some path $w' \in (\Sigma - \{r_l\})^* r_l^* (\Sigma - \{r_l\})^*$. ■

Remark 1

As seen in the proof, the requirement that all the coefficients $k_{l,j}$ should be nonnegative is unnecessarily strong. It is enough that $\vartheta_{r_j}(\overline{x}) \Rightarrow \vartheta_{r_j}(\overline{x} r_l)$ holds for every clause $r_j \in \Sigma$, which means that r_l preserves all the constraints of Σ.

Remark 2

It is clear that the languages involved in the right-part of the equivalence in proposition 2, viz. $(\Sigma - \{r_l\})^*$ and r_l^*, are of lower dimension than Σ provided that Σ contains more than one clause. (If Σ contains only one clause, say r_1, then the program is elementary and, as already pointed out, the relation $\overline{x} \xrightarrow{r_1^*} \overline{x}'$ is characterizable as an existentially quantified Presburger formula.)

Remark 3

Consider an elementary program Σ of dimension $(1, n)$. It means that all the columns of its incrementation matrix are null except one, say the h-th column. So the l-th row is monotonic (increasing if $k_{l,h} \geq 0$, or decreasing if $k_{l,h} \leq 0$), for any $1 \leq l \leq n$. Therefore one can apply the monotonic rule, thus decomposing program Σ into $\{r_l\}$ and $\Sigma - \{r_l\}$. For the same reasons, the monotonic rule applies again to the latter program $\Sigma - \{r_l\}$. By iteratively applying the rule, one can thus decompose the reachability problem via Σ^* into reachability problems via r_1^*, r_2^*, ..., r_n^*. It follows that one can characterize the reachability problem via Σ^* in the existential fragment of Presburger arithmetic.

Other monotonic decomposition rules are given in appendix A.

4.3. Cyclic decomposition rule

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

$$
\left.\begin{array}{llll}
\bullet \ \ldots \ \bullet & + & \bullet \ \ldots \ \bullet & \cdots \quad : r_1 \\
 & \vdots & & \\
\bullet \ \ldots \ \bullet & + & \bullet \ \ldots \ \bullet & \cdots \quad : r_l \\
+ \ \ldots \ + & -1 & + \ \ldots \ + & x_j > 0 \ : r_{l+1} \\
 & \vdots & & \\
+ \ \ldots \ + & -1 & + \ \ldots \ + & x_j > 0 \ : r_n \\
 & x_j & &
\end{array}\right\} \begin{array}{l} R' \\ \\ \\ \\ R \end{array}
$$

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 $k_{i,j}$ $(l+1 \leq i \leq n)$ 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.

Remark 4
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: $x_j > 0 \ \wedge\ \overline{x} \xrightarrow{R'R} \overline{x}' \ \Rightarrow\ \overline{x} \xrightarrow{RR'} \overline{x}'$.

Remark 5
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 3 *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

A

$$x_j \geq 0 \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Rightarrow$$

$$\overline{x} \xrightarrow{R^* R'^*} \overline{x}'$$
$$\vee$$
$$\exists \overline{x}'' : \ \overline{x} \xrightarrow{R^*} \overline{x}'' \xrightarrow{\Sigma^*} \overline{x}' \wedge x''_j = 0$$

B

$$x_j = 0 \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Rightarrow$$

$$\overline{x} \xrightarrow{\left(\bigcup_{r_i \in R'} r_i R^{k_{i,j}}\right)^* \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right) R'^*} \overline{x}'$$

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

C

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

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

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

Before proving this proposition, let us stress that part **C** of the 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.

Proof: The first statement, **A**, of the proposition states that any point $\overline{x}'$ reachable from a point $\overline{x}$ such that $x_j \geq 0$, is reachable either by a path consisting of a sequence of applications of clauses of R only followed by a sequence of applications of clauses of R' only, or $\overline{x}'$ is reachable via a point $\overline{x}''$ (with $x''_j = 0$), which is itself reachable from $\overline{x}$ by a sequence of applications of clauses of R only. We prove this by induction on the length n of the paths. The case when $n = 0$ is trivial. The induction hypothesis is the following implication: For all $v \in \Sigma^*$ such that $|v| < n$, $(x_j \geq 0 \wedge \overline{x} \xrightarrow{v} \overline{x}' \;\Rightarrow\; (\overline{x} \xrightarrow{v'} \overline{x}' \;\vee\; \exists \overline{x}'' : \; \overline{x} \xrightarrow{v''} \overline{x}'' \xrightarrow{v'''} \overline{x}' \wedge x''_j = 0))$, for some $v' \in R^* R'^*$, $v'' \in R^*$, $v''' \in \Sigma^*$ such that $|v'| = |v''v'''| = |v|$. Suppose now that $x_j \geq 0 \wedge \overline{x} \xrightarrow{w} \overline{x}'$ hold for some $\overline{x}, \overline{x}'$ and $w \in \Sigma^*$ such that $|w| = n$, and let us prove **D**: $(\overline{x} \xrightarrow{w'} \overline{x}' \;\vee\; \exists \overline{x}''' : \; \overline{x} \xrightarrow{w''} \overline{x}''' \xrightarrow{w'''} \overline{x}' \wedge x'''_j = 0)$ for some $w' \in R^* R'^*$, $w'' \in R^*$ and $w''' \in \Sigma^*$ such that $|w'| = |w''w'''| = |w|$. If no R-clause appears in w, then $w \in R'^*$ so clearly $w \in R^* R'^*$, and **D** follows by choosing w' as w. Otherwise some clause of R must occur in w for the first time. Then $w = w_1 r_i w_2$ for some $w_1 \in R'^*$, $r_i \in R$ and $w_2 \in \Sigma^*$. If $x_j = 0$ we choose $\overline{x}''' = \overline{x}$, $w'' = \varepsilon$ and $w''' = w_1 r_i w_2$, which again proves **D**. Therefore assume $x_j > 0$. By precondition 2, all the clauses of R' make x_j increase, so $x_j > 0$ is invariant for all the paths in R'^*. By repeated use of precondition 1, $w_1 r_i$ may then be replaced by some $r'_i w'_1$ such that $r'_i \in R$, $w'_1 \in R'^*$ and $|w'_1| = |w_1|$, so $\overline{x} \xrightarrow{r'_i} \overline{x}'' \xrightarrow{w'_1 w_2} \overline{x}'$ holds for some $\overline{x}''$. By precondition 3, all the clauses in R decrease x_j by one, so either $x''_j = 0$, in which case we choose w'' as r'_i and w''' as $w'_1 w_2$ for proving **D**, or $x''_j > 0$ still holds. Since $|w'_1 w_2| < |w|$, by the induction hypothesis, $\overline{x} \xrightarrow{v'} \overline{x}' \;\vee\; \exists \overline{x}''' : \; \overline{x}'' \xrightarrow{v''} \overline{x}''' \xrightarrow{v'''} \overline{x}' \wedge x'''_j = 0$, holds for some $v' \in R^* R'^*$, $v'' \in R^*$, $v''' \in \Sigma^*$ and $|v'| = |v''v'''| = |w'_1 w_2|$, and therefore $\overline{x} \xrightarrow{r'_i v'} \overline{x}' \;\vee\; \exists \overline{x}''' : \; \overline{x} \xrightarrow{r'_i v''} \overline{x}''' \xrightarrow{v'''} \overline{x}' \wedge x'''_j = 0$. Thus **D** holds, since $r'_i v' \in R^* R'^*$, $r_i v'' \in R^*$ and $|r_i v'| = |r_i v'' v'''| = |w|$. The slightly stronger result that $|w'| = |w''w'''| = |w|$, will be used below.

The second statement, **B**, says that if $\overline{x}'$ is reachable from some point $\overline{x}$ such that $x_j = 0$, then $\overline{x}'$ is reachable by a sequence of repeated cycles $r_i R^{k_{i,j}}$, where $r_i \in R'$, possibly followed by a prefix $r_i R^{0<*<k_{i,j}}$ of a cycle and finally by a sequence of applications of clauses of R' only. It is obvious that the paths $r_i R^{k_{i,j}}$ keep $x_j = 0$ invariant since r_i increases x_j by $k_{i,j}$, and all clauses of R decreases x_j by one, so r_i followed by $k_{i,j}$ applications of R-clauses sums up to zero. The statement is proved by induction. Again the base case when $n = 0$ is trivial. The induction hypothesis is the following implication: for all $v \in \Sigma^*$ such that $|v| < n$, $\quad x_j = 0 \wedge \overline{x} \xrightarrow{v} \overline{x}' \;\Rightarrow\; \overline{x} \xrightarrow{LR'^*} \overline{x}'$, where $L = \left(\bigcup_{r_i \in R'} r_i R^{k_{i,j}}\right)^* \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right)$. Suppose that $x_j = 0 \wedge \overline{x} \xrightarrow{w} \overline{x}'$ hold for some $\overline{x}$, $\overline{x}'$ and $w \in \Sigma^*$ such that $|w| = n$, and let us prove $\overline{x} \xrightarrow{LR'^*} \overline{x}'$. Since $x_j = 0$, by precondition 4, no clause of R can be applied, so the first clause application must be some $r_i \in R'$, and therefore $w = r_i w_1$ for some $w_1 \in \Sigma^*$. Thus $x_j = 0 \wedge \overline{x} \xrightarrow{r_i} \overline{x}'' \xrightarrow{w_1} \overline{x}'$ holds for some $\overline{x}''$. If $k_{i,j} = 0$, then $x''_j = 0$. Since $|w_1| < |w|$, by the induction hypothesis, $\overline{x}'' \xrightarrow{LR'^*} \overline{x}'$

holds, so $\overline{x} \xrightarrow{r_i L R'^*} \overline{x}'$. This proves $\overline{x} \xrightarrow{LR'^*} \overline{x}'$, since $r_i L \subseteq L$. Therefore assume $k_{i,j} > 0$, in which case $x''_j > 0$ must hold. But by the proof of case **A** of the proposition, if $x''_j > 0 \wedge \overline{x}'' \xrightarrow{w_1} \overline{x}'$ holds, then either **E1**: $\overline{x}'' \xrightarrow{w'_1} \overline{x}'$ or **E2**: $\exists \overline{x}''' : \overline{x}'' \xrightarrow{w''_1} \overline{x}''' \xrightarrow{w'''_1} \overline{x}' \wedge x'''_j = 0$ must hold for some $w'_1 \in R^* R'^*$, $w''_1 \in R^*$ and $w'''_1 \in \Sigma^*$ such that $|w'_1| = |w''_1 w'''_1| = |w_1|$. Assume that **E2** holds. Then $|w''_1| = k_{i,j}$ must hold and, since $|w'''_1| < |w|$, by the induction hypothesis, $\overline{x}''' \xrightarrow{LR'^*} \overline{x}'$, so $\overline{x} \xrightarrow{r_i w''_1 L R'^*} \overline{x}'$, which proves $\overline{x} \xrightarrow{LR'^*} \overline{x}'$, since $r_i w''_1 L \subseteq L$. Suppose now that **E2** does *not* hold. By **E1**, $w'_1 = uu'$ for some $u \in R^*$ and $u' \in R'^*$. Furthermore, $|u| < k_{i,j}$ must hold, since otherwise w''_1 could be chosen as u, and **E2** would hold. Therefore $r_i u u' \in \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right) R'^* \subseteq LR'^*$, and again, $\overline{x} \xrightarrow{LR'^*} \overline{x}'$ holds.

The third statement, **C**, follows by simply combining **A** and **B**, and by noting that if $x_j < 0$, by precondition 4, only R'-clauses can be applied. Either we reach the end point, or we reach some point where $x_j \geq 0$, and then cases **A** and **B** of the proposition apply. ■

Remark 6:

In the special case where the constraints of R-clauses are atomic (i.e., all equal to $x_j > 0$), it is easy to show that the application of R-clauses is commutative. Therefore, we have for all r_i in R'

$$\overline{x}'' \xrightarrow{r_i R^k} \overline{x}''' \;\Rightarrow\; \overline{x}'' \xrightarrow{r_i \bigcup_{m_1+m_2+\ldots+m_{n-l}=k} r_{l+1}^{m_1} r_{l+2}^{m_2} \cdots r_n^{m_{n-l}}} \overline{x}'''$$

Hence we need not consider all the paths of $r_i R^k$, but only those of the form $r_i r_{l+1}^{m_1} r_{l+2}^{m_2} \cdots r_n^{m_{n-l}}$, where $m_1 + m_2 + \ldots + m_{n-l} = k$. (Actually, the ordering on $r_{l+1}, r_{l+2}, \ldots r_n$ is arbitrary.)

Remark 7:

In the special case where the constraints of R-clauses are atomic (i.e., all equal to $x_j > 0$), let us also notice that, for all clause $w \in r_i R^k$, the associated constraint $\vartheta_w(\overline{x}'')$ is equal to $\vartheta_{r_i}(\overline{x}'')$: This is trivial if $k = 0$; in the case where $k > 0$, constraint $\vartheta_w(\overline{x}'')$ is equal to $\vartheta_{r_i}(\overline{x}'') \wedge x''_j + k > 0 \wedge \ldots \wedge x''_j + 1 > 0$, and reduces to ϑ_{r_i} because x''_j is equal to 0 (see statement **C** of proposition 3). In other words, one can always drop the constraints relevant to x_j within clauses of $r_i R^k$. One can see that x_j-constraints can be dropped also in the general case where constraints of R-clauses are not atomic.

EXAMPLE: Consider the matrix in example 4, representing the program for the protocol of example 3.4. Let us in proposition 3 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 3 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 3.C, we have:

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

$$\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$$
$$\vee$$
$$\exists \overline{x}'' : \; \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*} \overline{x}'' \wedge$$
$$\overline{x}'' \xrightarrow{(r_1+r_2+r_3 r_5+r_3 r_6+r_4 r_5+r_4 r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}' \wedge x''_5 = 0$$

and all the paths in $(r_1 + r_2 + r_3 r_5 + r_3 r_6 + r_4 r_5 + r_4 r_6)^*$ keep $x_5 = 0$ invariant. The matrix M' of the program corresponding to the set of clauses $\{r_1, r_2, r_3, r_5, r_3 r_6, r_4 r_5, r_4 r_6\}$ is shown below:

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

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

□

5. Comparison with Related Work

5.1. 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 $x_5 > -1$ (*cf.* Remark 7). 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 2. 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 in (Berthelot, 86). Our version of the cyclic decomposition can thus be seen as a variant of Berthelot's post-fusion rule. Berthelot also defined some other transformations like *pre-fusion*. It is possible to give in our framework a counterpart also for this transformation (see appendix A).

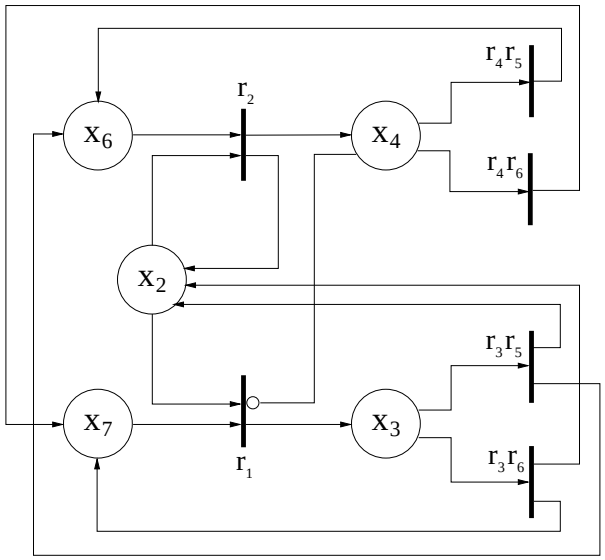


Figure 2

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 $\mathcal{Z}$-counters where variables take their values on $\mathcal{Z}$ (instead of $\mathcal{N}$ as in the case of Petri nets). This allows us in particular to encode 0-tests as already seen. In section 7 we will see that our cyclic decomposition rule can also, under certain conditions, be generalized one step further by allowing R-transitions to pick up more than one token from place x_j. (For rules $r_i \in R$, coefficients $k_{i,j}$ will be allowed to be less than -1.)

5.2. Comparison with related work in constraint databases

Our decomposition can be seen as a means of eliminating redundant paths (proof derivations) leading from a fact to another fact. This issue of eliminating redundancy during bottom-up execution of Datalog or logic programs (with constraints) has given rise to two different kind of methods: static methods and dynamic ones. In the static approach, basic rules of transformation are applied to the program itself in order to narrow its bottom-up tree of derivations. This is applicable when the program satisfies certain properties for which sufficient syntactic criteria exist: e.g., boundedness (Naughton, 87), commutatitivity (Ramakrishnan, 89), splittability (Nilsson, 82)(Lassez, 83). In the dynamic approach, redundant derivations are eliminated during the execution of the program. For example, (Vieille, 89) discusses the run-time detection and elimination of redundant subgoals and redundant parts of SLD-derivation trees. In (Helm, 89) redundant derivations are removed during bottom-up execution by, first, unfolding the original program (see (Tamaki, 84)) according to a strategy defined by a control language, then eliminating redundant

unfolded clauses. In the dynamic approach, the detection of redundancy basically relies on various enhancements of the classical notion of "subsumption" (*cf.* (Loveland, 78)(Maher, 88)). Our work here belongs to the static approach. One of the decomposition rules of our system (see the rule of "stratification", proposition 5, appendix A) is thus a commutativity rule in the sense of (Ramakrishnan, 89). The main originality of our system lies in the subset of cyclic rules, which do not simply rearrange clauses of the original program, but create new clauses by "fusion" of old ones. This fusion can be interpreted as a restricted form of unfolding as in the work of (Helm, 89), but here, the strategy of unfolding is fixed by the rule, and its correctness always guaranteed without need for dynamic tests of subsumption. Let us finally mention that dynamic tests of subsumption can still be easily integrated within our method. For example, we will see in section 8 how tests of invariance (analogous to the subsumption tests used in bottom-up evaluation of constraint databases (Maher, 93)) are added to the basic decomposition procedure in order to optimize the construction of the least fixed-point.

6. Application to BPP-Nets

Since programs with $\mathcal{Z}$-counters are Turing equivalent it is clear that the decomposition process cannot always succeed. Even for Petri nets it is known that nets with more than four places, in general do not have a a reachability set expressible in linear arithmetic (Hopcroft, 79). In this section we consider a subclass of Petri nets for which the decomposition process is guaranteed to succeed in generating a flat language.

Recently an interesting subset of Petri nets has been introduced and investigated: BPP-nets. A Petri net is a BPP-net if every transition has at most [2] 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 (Christensen, 93); the reachability problem for BPP-nets is NP-complete (Esparza, 95). 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 $x_h > 0$. We call such a clause r_i a *BPP-clause*. Let us assume given a BPP-net Σ (i.e., a set of BPP-clauses), and consider the following property

Prop(Σ): Σ^* can be decomposed into a sequence $L_1 ... L_s$ such that, for all $1 \leq i \leq s$, the language L_i is either finite, or of the form w_i^* for some path w_i, or of the form Σ_i^* for some BPP-net Σ_i.

By iterative application of this proposition, one generates eventually a *flat* decomposition of the given BPP-net Σ. (The process terminates because all our rules of decomposition transform a program into programs of lower dimension.) Let us

now prove Prop(Σ).

Proof: If there exists a clause r_i in Σ such that all the coefficients of its head are nonnegative, the monotonic (increasing) decomposition rule is applied, and Σ^* is decomposed into $(\Sigma-\{r_i\})^* r_i^* (\Sigma-\{r_i\})^*$. If there are still such clauses in $\Sigma-\{r_i\}$, we apply again the monotonic rule, and so on until one gets a sequence made only of expressions of the form r_j^* and expressions of the form Σ'^*, where Σ' denotes the subset of clauses of Σ having at least one negative coefficient. By assumption, every clause of Σ' must then contain *exactly one* negative coefficient (equal to -1) and its constraint must be atomic. In order to prove Prop(Σ), it then suffices to prove Prop(Σ'). Assume that Σ' is nonempty (otherwise, the property is trivial), and let us show that the cyclic decomposition rule applies to Σ'. We have to determine which sets of rules to take as for R, R' and which variable to take as for x_j in order to apply proposition 3. As for x_j we choose a variable such that column j of the matrix contains an element equal to -1 (which must exist). As R we take all the clauses r_i such that $k_{i,j} = -1$, and as R' we take $\Sigma' - R$.

Let us show that R can be decomposed under a flat form. Since all the clauses $r_i \in R$ have the same atomic constraint $x_j > 0$ and all the coefficients $k_{i,j}$ are equal (to -1), one easily sees that all the rules in R commute as required for the stratification decomposition of proposition 5 (see appendix A). So by repeatedly applying this decomposition, R^* can be flattened. (Actually, if R is made of l clauses $r_{i_1}, .., r_{i_l}$, R^* can be decomposed into the flat form $r_{i_1}^* ... r_{i_l}^*$.)

If R' is empty, then Σ' is R, and can thus be put under a flat form. Assume therefore that R' is nonempty. Thus, for every $r_i \in R'$ we have $k_{i,j} \geq 0$, and therefore conditions $1, 1', 2, 3, 4$ of the specialized case of section 4.2 are satisfied, so the cyclic decomposition applies. By equivalence **C** of proposition 3, reachability by Σ'^* is reduced to reachability as:

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

$$\overline{x} \xrightarrow{R'^* R^* R'^*} \overline{x}'$$

$$\vee$$

$$\exists \overline{x}'' : \; \overline{x} \xrightarrow{R'^* R^*} \overline{x}'' \xrightarrow{\left(\bigcup_{r_i \in R'} r_i R^{k_{i,j}}\right)^* \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right) R'^*} \overline{x}' \;\wedge\; x_j'' = 0$$

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

Clearly R' is still of the form corresponding to a BPP-net. Besides R^* can be put under a flat form, and $\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}$ is finite. In order to achieve the proof of Prop(Σ'), it then suffices to show that $\bigcup_{r_i \in R'} r_i R^{k_{i,j}}$ corresponds to a BPP-net, that is: for any clause $w \in \bigcup_{r_i \in R'} r_i R^{k_{i,j}}$, all the coefficients $k_{w,j}$ are nonnegative except (maybe) one, which is equal to -1; besides, if such a negative coefficient,

say $k_{w,h}$, exists, the constraint of w is atomic and equal to $x_h > 0$. Let us consider those rows of matrix Σ' which are involved in $\bigcup_{r_i \in R'} r_i R^{k_{i,j}}$:

$$\begin{array}{cccccccccccccc} + & \ldots & + & k_{i,h} & + & \ldots & + & k_{i,j} & + & \ldots & + & \vartheta_{r_i} & : r_i \in R' \\ + & \ldots & + & + & + & \ldots & + & -1 & + & \ldots & + & x_j > 0 & \Big\} R \\ & & & \vdots & & & & & & & & & \\ + & \ldots & + & + & + & \ldots & + & -1 & + & \ldots & + & x_j > 0 & \Big\} \\ & & & x_h & & & & x_j & & & & & \end{array}$$

By composition of these rows, any clause w in $\bigcup_{r_i \in R'} r_i R^{k_{i,j}}$, has a vector of coefficients $\overline{k}_w$ of the form: $\langle +, ..., +, l_h, +, ..., +, 0, +, ..., + \rangle$ with $l_h \geq k_{i,h}$, and a constraint $\vartheta_w(\overline{x})$ equal to $\vartheta_{r_i}(\overline{x})$ (see remark 7). If $k_{i,h}$ is nonnegative, then all the coefficients of w are non negative (since $l_h \geq k_{i,h}$). If $k_{i,h} = -1$, then ϑ_{r_i} is of the form $x_h > 0$, and so is ϑ_w (since $\vartheta_w = \vartheta_{r_i}$); besides all the coefficients of w are nonnegative (except perhaps, l_h, which may be equal to -1). In any case, clause w is a BPP-clause, and $\bigcup_{r_i \in R'} r_i R^{k_{i,j}}$ corresponds to a BPP-net. This completes the proof of Prop(Σ'), hence of Prop(Σ). ■

Thus, for BPP-nets, the decomposition process is guaranteed to terminate successfully and one obtains an existentially quantified Presburger arithmetic formula having $\overline{y}$ as a free variable for characterizing the fact that $\overline{y}$ belongs to the reachability set. This yields a new proof of the fact that the reachability set for BPP-nets is a semilinear set (Esparza, 95). Note that Esparza's proof makes use of the notion of "siphon", and is completely different from our method. Note also that our result is actually more general since our decomposition succeeds for BPP-nets without any assumption on the initial markings: our decomposition process shows that the 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 (Ginsburg, 66)) while the result of Esparza states that $\{\overline{x}' \mid \overline{a}^0 \xrightarrow{\Sigma^*} \overline{x}'\}$ is a semilinear set, for any tuple of constants $\overline{a}^0$).

Remark 8:

The requirement that, in a BPP-clause r_i, the constraint should be *atomic* (equal to $x_h > 0$) in case there is a negative coefficient $k_{i,h}$, is essential here to our proof of semi-linearity of the reachability set. Otherwise, it is not possible to apply the postfusion rule (because r_i does not commute in general with R-clauses). Actually, the result of semi-linearity does not extend to programs with non atomic constraints: (Hopcroft, 79) gives a vector addition system (VAS) corresponding to a Datalog program with $\mathcal{Z}$-counters made of 5 clauses having nonnegative coefficients (except one equal to -1) and *non* atomic constraints, for which the reachability set is *not* semi-linear.

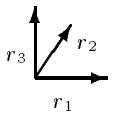
7. A Generalized Form of the Decomposition Cyclic Rule

The cyclic decomposition rule presented above, can be given a more general formulation, which makes it applicable even when the coefficient $k_{i,j}$ of R-rule r_i is

less than -1. We illustrate this general formulation by considering a 3-clauses program of a typical form and vizualizing its associated least fixpoint. The program is defined by the base case vector $\langle 30, 19, -57\rangle$ and the incrementation matrix:

$$\begin{array}{rrrll} -2 & -1 & 3 & x_1 > 0 & : r_1 \\ -1 & -2 & 4 & x_2 > 0 & : r_2 \\ 4 & 3 & -7 & x_3 > 0 & : r_3 \\ x_1 & x_2 & x_3 & & \end{array}$$

Its least fixpoint is represented in figure 3, under the form of the set of all its applicable paths. All horizontal (resp. vertical, transversal) segment of a path corresponds to the application of the first (resp. second,third) recursive rule. The orientation of the figures in terms of r_1, r_2 and r_3 is:



Let us choose x_j to be x_3, R' to be $\{r_1, r_2\}$ and R to be $\{r_3\}$. A priori, propo-

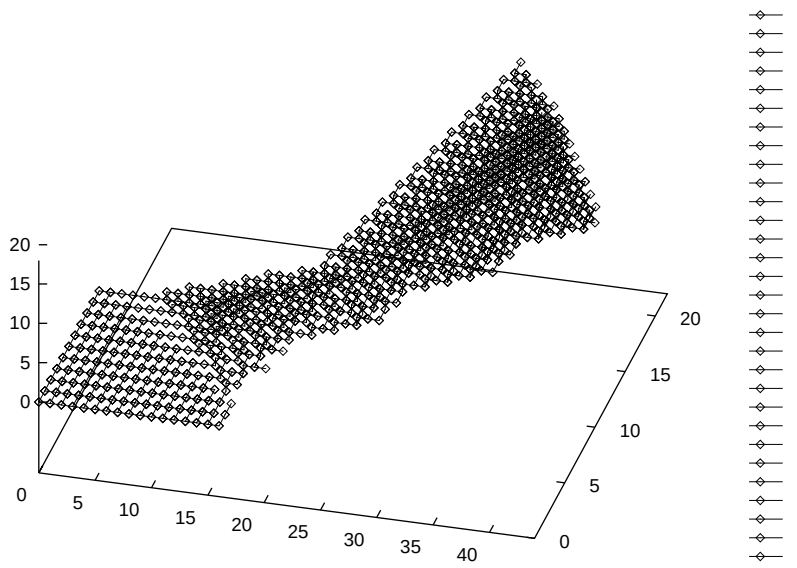


Figure 3

sition 3 does not apply because $k_{3,3}$, viz. -7, is not equal to -1. We are going to explain however that an analogous decomposition applies, and that, similarly to what part **C** of proposition 3 says, we have: $\langle 30, 19, -57\rangle \xrightarrow{(r_1+r_2+r_3)^*} \overline{x}' \quad \Rightarrow$

$\langle 30, 19, -57\rangle \xrightarrow{(r_1+r_2)^* r_3^* L (r_1+r_2)^*} \overline{x}'$ where L is a counterpart of the sequences of cyclic sequences $(\bigcup r_i R^{k_i})^* (\epsilon + \bigcup r_i R^{0<*<k_i})$.

Compare the language expression $(r_1 + r_2)^* r_3^* L (r_1 + r_2)^*$ with figure 3. The lower left part of the figure is a planar area where 2 rules only are applicable (one coordinate, viz. x_3, remains always less than or equal to zero), and is therefore included into a $\{r_1, r_2\}$-plane, which corresponds to the initial sublanguage $(r_1 + r_2)^*$. After a while, $x_3 > 0$ becomes true, and a number of transversal moves r_3 apply, which is captured by the sublanguage r_3^*. As is seen in the figure, the r_3-moves soon cease. After the last application of r_3, it must hold that $0 \geq x_3 > -7$ (since r_3 is not applicable, but was applicable immediately before, and the application of r_3 makes x_3 decreased by 7). At this point, only r_1 and r_2 can be applied. Since r_1 makes x_3 increase by 3, and r_2 by 4, we must have: $4 \geq x_3 > 0$, when x_3 becomes strictly greater than zero for the first time. Now, in keeping with the firing strategy of proposition 3, clause r_3 should be fired as soon as it is applicable. In the figure, the set of points reached by such a strategy is the "ceiling" of the cone (we move upwards as soon as we can). [3] The coordinate x_3 of a reordered path is thus led to take cyclically 11 values, those between 4 and -6. These values can be considered as *states* of a deterministic finite state automaton defining the language L of reordered paths. The transitions of such an automaton are completely defined by our strategy of clause firing, which gives priority to clause r_3 whenever it is applicable (i.e., when $x_3 > 0$). From any state, $4 \geq x_3 > 0$, there is only one arrow going out, labeled r_3, and the next state is $x_3 - 7$. From any state $0 \geq x_3 > -7$, there are two arrows going out, one labeled r_1 for which the next state is $x_3 + 3$, and one labeled r_2 for which the next state is $x_3 + 4$. The automaton is shown in figure 4. The construction is identical to that of (Clausen,

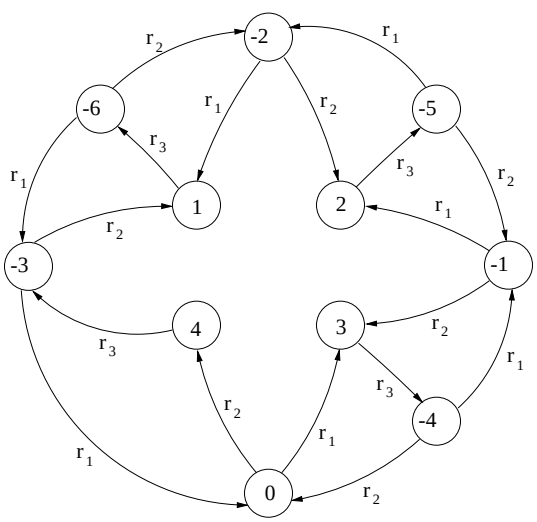


Figure 4

89) for solving the linear diophantine equation:

$$3m_1 \ + \ 4m_2 \ - \ 7m_3 \ = \ 0$$

which expresses the fact that any path consisting of m_1, m_2 and m_3 applications of r_1-,r_2- and r_3-clauses, respectively, lets x_3 invariant. It is easy to see that every cycle in the automaton yields a solution to the equation above.

Let us denote by $L_{s,t}$ the language of paths leading from state s to state t. It should be clear now that one can always reorder paths as follows:

$$\langle 30, 19, -57\rangle \xrightarrow{(r_1+r_2+r_3)^*} \overline{x}' \quad \Rightarrow \quad \langle 30, 19, -57\rangle \xrightarrow{(r_1+r_2)^* r_3^* L (r_1+r_2)^*} \overline{x}'$$

where L is $\bigcup_{4\geq s,t>-7} L_{s,t}$. This is because, first, as already seen, we use $(r_1+r_2)^*$ paths or r_3^* until one reaches a state $4 \geq s > -7$ in the automaton (that is, $x_3 = s$). Then the clauses are fired according to the strategy defined by the automaton until no more r_3 clauses are still to be fired. From that point on one walks in the $(r_1 + r_2)^*$-plane, leaving the automaton at some state $4 \geq t > -7$ (that is, $x_3 = t$ and $x_3 r_i > 4$ for $i = 1, 2$). In figure 3 this means following the "ceiling" of the cone, and then "filling it up" with $\{r_1, r_2\}$-planes. This informal explanation of the example can be turned into a formal proof of the following generalization of proposition 3.

PROPOSITION 4 *Let $R, R' \subseteq \Sigma$ be a set of clauses such that $\Sigma = R \uplus R'$, and let x_j be a variable and c some fixed constant such that:*

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

Then there exists a finite set of languages $L_{s,t}$, with $b \geq s, t > a$, where $a = \min\{k_{i,j} + c \mid r_i \in R\}$ and $b = \max\{k_{i,j} + c \mid r_i \in R'\}$, such that:

A

$$x_j > c \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Rightarrow$$

$$\overline{x} \xrightarrow{R^* R'^*} \overline{x}'$$
$$\vee$$
$$\exists \overline{x}'' : \ x_j > c \wedge \overline{x} \xrightarrow{R^*} \overline{x}'' \xrightarrow{\Sigma^*} \overline{x}' \wedge b \geq x''_j > c$$

B

$$\forall b \geq s > a : \ \left(\begin{array}{l} x_j = s \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Rightarrow \\ \\ \qquad \overline{x} \xrightarrow{\left(\bigcup_{b>t>a} L_{s,t}\right) R'^*} \overline{x}' \end{array} \right)$$

C

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

$$\overline{x} \xrightarrow{R'^* R^* \left(\bigcup_{b>s,t>a} L_{s,t}\right) R'^*} \overline{x}'$$

As can be seen on figure 4, the languages $L_{s,t}$ are in general not of the form $L_1 L_2 \cdots L_u$ where L_i is either finite or of the form Σ_i^* (with Σ_i finite), but may contain nested '*'. For example the expression $(r_1 r_3 (r_1 r_2 r_3)^* r_2)^*$ is a subset of the language $L_{0,0}$, while $(r_1 r_3 r_2 + r_1 r_2 r_3)^*$ is not. This means that proposition 4 may not in general be applied iteratively. However, by applying other decompositions such as monotonic rules, one can sometimes retrieve a language that can be expressed under such a "flat" form (without nesting of '*'). For a program with 3 recursive clauses and atomic constraints, as the one above, whose matrix has the general form:

$$\begin{array}{ccccl} \bullet & \bullet & + & x_1 > 0 & : r_1 \\ \bullet & \bullet & + & x_2 > 0 & : r_2 \\ + & + & - & x_3 > 0 & : r_3 \\ x_1 & x_2 & x_3 & & \end{array}$$

we have shown that such a decomposition is *always* possible, which allows to solve the problem of the arithmetical characterization of the least fixed-point (see (Fribourg, 95)).

We can look back at the results stated in proposition 3, and interpret them as a special case of the above automaton-based construction. Under the conditions of proposition 3, the constant $a = \min\{k_{i,j} + c \mid r_i \in R\}$ is equal to -1. So the states of the automaton range here from 0 to b. For each nonnull state, there is one outgoing R-arc and some entering R'-arcs. For the null state $s = 0$, there is one entering R-arc and some outgoing R'-arcs. The reordered paths, as defined by part **C** of proposition 3, can now be constructed, using this specialized automaton, as illustrated on figure 5. Figure 6 gives a geometrical interpretation of the fact all the cycles closely follow the hyper plane $x_j = 0$.

8. Compilation into Arithmetic

We have an experimental implementation in SICSTUS-PROLOG currently containing seven decomposition rules, two of which are cyclic (*cf.* appendix A). Besides the decomposition module, it contains a theorem prover for Presburger arithmetic based on the decision procedure of (Boudet, 96). 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 (Boudet, 96)).

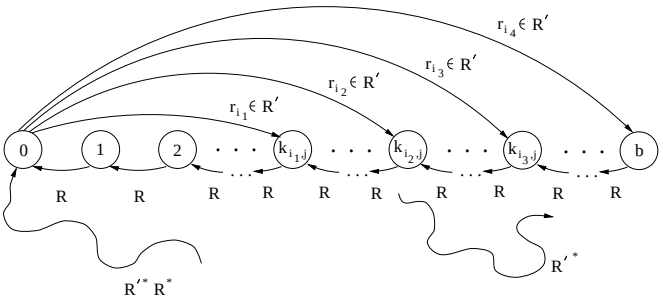


Figure 5

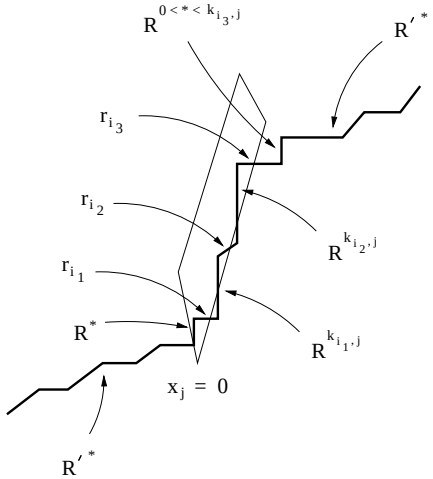


Figure 6

For the readers-writers protocol of figure 1, 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 appendix A).

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 lfp : $\{\overline{x}' \mid \exists \overline{x} : B(\overline{x}) \wedge \overline{x} \xrightarrow{w_1^* w_2^* \cdots w_{51}^*} \overline{x}'\}$. 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: $\overline{x} \in \text{lfp} \Leftrightarrow \xi_{51}(\overline{x})$. 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 lfp 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., (Maher, 93)). In the present case of the 51 strings long expression, the least fixed-point is thus reached after only 4 steps, and is thus given by

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

The corresponding arithmetical expression is (see appendix B):

$$\begin{aligned} \text{lfp} \;\equiv\; \xi_4(\overline{x}) \;\Leftrightarrow\; & x_1 = 1 - x_4 \wedge \\ & ((x_2 = 1 \wedge x_3 = 0 \wedge x_4 \geq 0) \vee \\ & \;(x_2 = 0 \wedge x_3 = 1 \wedge x_4 = 0)) \wedge \\ & x_5 \geq 0 \wedge x_6 \geq 0 \wedge x_7 \geq 0 \wedge \\ & x_3 + x_4 + x_5 + x_6 + x_7 = q \end{aligned}$$

It is easy to see that the mutual exclusion property, $x_3 = 0 \vee x_4 = 0$, holds. Our program constructs the arithmetical form of the least fixed-point in 1.1 second, and proves the property in 0.5 second.

Details on our 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 (Fribourg, 97).

9. Conclusion

We have developed a decompositional approach for computing the least fixed-points of Datalog programs with $\mathcal{Z}$-counters. As an application we have focused on the computation of reachability sets for Petri nets. We have thus related some unconnected topics such as Berthelot's transformation rules and Esparza's semilinearity result for the reachability set of BPP-nets. We have also shown how these results can be extended in several directions (BPP-nets with parametric initial markings, post-fusion rule for Petri nets with input arcs picking up more than one token). Our system implementation gives already promising results, as illustrated here on the readers-writers protocol. Other experimental results in the field of parametrized protocols and manufacturing systems are presented in (Fribourg, 97).

Acknowledgments

Special thanks are due to Alain Finkel for providing us with many useful informations on Petri nets. We thank also the referees for their helpful suggestions.

Appendix A

We present here some other decomposition rules used in our system in addition to the rules of monotonic clause and cyclic postfusion. The first rule is called *stratification*. Exploiting some commutativity property (*cf.* (Ramakrishnan, 89)), it states that some clauses can be applied before all the others.

PROPOSITION 5 *Let $R, R' \subset \Sigma$ be sets of clauses such that $\Sigma = R \uplus R'$, and such that $\overline{x} \xrightarrow{R'R} \overline{x}' \Rightarrow \overline{x} \xrightarrow{RR'} \overline{x}'$. Then we have: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{R^*R'^*} \overline{x}'$*

Note that the condition $\overline{x} \xrightarrow{R'R} \overline{x}' \Rightarrow \overline{x} \xrightarrow{RR'} \overline{x}'$ reduces to check a finite set of inequalities among constants. The second decomposition rule is called *monotonic guard* and comes in two versions: *increasing* and *decreasing*. It is essentially "constraint pushing" (Srivastava, 92) and applies when there is a single-signed column. We present here only the decreasing version.

PROPOSITION 6 *Let $R \subseteq \Sigma$ be a set of clauses and let x_j be a variable such that:*

1. $\forall r_i \in \Sigma : \; k_{i,j} \leq 0$
2. $\forall r_i \in R : \; \vartheta_{r_i}(\overline{x}) \Rightarrow x_j > c$ *for some fixed constant c.*

Then: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \Leftrightarrow (\overline{x} \xrightarrow{(\Sigma - R)^*} \overline{x}' \vee \exists \overline{x}'' : \; \overline{x} \xrightarrow{\Sigma'^*} \overline{x}'' \xrightarrow{\Sigma(\Sigma - R)^*} \overline{x}' \wedge x''_j > c)$ where Σ' is obtained from Σ by removing all constraints of the form $x_j > c$ from every clause in R.

Another cyclic decomposition rule is *pre-fusion*. In the post-fusion decomposition the set Σ was divided into the sets R and R'. Intuitively, R "consumes" a resource while R' is a "supplier". The idea of the post-fusion strategy was to "consume" as soon as possible: clauses of R were fired as soon as their constraints were satisfied, thus having high priority. In contrast, the idea of pre-fusion is to focus on R'-clauses and to "delay" their firing as long as possible. For post-fusion it was not necessary to distinguish the rules $r_i \in R'$ that strictly increase the variable x_j, from those that let x_j invariant (that is, distinguish $k_{i,j} > 0$ from $k_{i,j} = 0$). For pre-fusion, instead of R' we consider two sets R' and R'' where R' are now those rules that strictly increase the variable and R'' are those that let it invariant. (Besides, coefficients $k_{i,j}$ of clauses $r_i \in R'$ must be equal to $+1$, which is a restriction w.r.t. post-fusion.) We wish to "delay" the application of R'-rules until immediately before the application of an R-rule. For this to be possible, R'-rules must be permutable with R''-rules that may occur between an R'-rule and an R-rule. This requirement is expressed by the first precondition of proposition 7 below. Condition 4 essentially says that R''-rules let the variable invariant. The rest of the preconditions are the same as for post-fusion.

PROPOSITION 7 *Let $R, R', R'' \subseteq \Sigma$ be disjoint sets of rules such that $\Sigma = R \uplus R' \uplus R''$, and let x_j be a variable such that:*

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

Then we have:

A

$$x_j \geq 0 \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Rightarrow$$

$$\overline{x} \xrightarrow{(R''+R)^* R'^*} \overline{x}'$$
$$\vee$$
$$\exists \overline{x}'' :\; \overline{x} \xrightarrow{(R''+R)^*} \overline{x}'' \xrightarrow{\Sigma^*} \overline{x}' \wedge x_j'' = 0$$

B

$$x_j = 0 \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Rightarrow$$

$$\overline{x} \xrightarrow{(R'R+R'')^* R'^*} \overline{x}'$$

where x_j is let invariant by all the paths in $(R'R + R'')^*$.

C

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

$$\overline{x} \xrightarrow{(R''+R)^*R'^*} \overline{x}'$$

$$\vee$$

$$\exists \overline{x}'' : \; \overline{x} \xrightarrow{R''^*R'^*+(R''+R)^*} \overline{x}'' \xrightarrow{(R'R+R'')^*R'^*} \overline{x}' \wedge x''_j = 0$$

where x_j is let invariant by all the paths in $(R'R + R'')^*$.

Appendix B

Let us compute the fixed-point of the program P' of example 3.4 from the language $L = r_5^* r_6^* r_2^* r_1^* \ldots$ of section 8, generated by our program. We get the sequence (making arithmetic simplifications at each step):

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

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.

Notes

1. Coefficients λ_i are found by solving the system $\langle \lambda_1, ..., \lambda_7 \rangle . M = \langle 0, ..., 0 \rangle$, where M is a matrix whose j-th column $(1 \leq j \leq 6)$ is vector $\overline{k}_{r_j}$.
2. The original definition states that every transition has *exactly* one input place, but it is convenient here to relax it somewhat.

3. This incidentally shows that the set of points in the "ceiling" of the figure must satisfy $4 \geq x_3 > -7$.

References

Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S. and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, Chichester,1995.

Baudinet, M., Niezette, M. and P. Wolper, "On the Representation of Infinite Temporal Data Queries". *Proc. 10th ACM Symp. on Principles of Database Systems*, Denver, 1991, pp. 280-290.

Berthelot, G., "Transformations and Decompositions of Nets". *Advances in Petri Nets*, LNCS 254, Springer-Verlag, 1986, pp. 359-376.

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.

Brams, G.W., *Réseaux de Petri: Théorie et Pratique*, Masson, Paris, 1983.

Chomicki, J. and T. Imielinski, "Temporal Deductive Databases and Infinite Objects". *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, 1988, pp. 61-81.

Christensen, S., *Decidability and Decomposition in Process Algebras.* Ph.D. Thesis, University of Edinburgh, CST-105-93, 1993.

Clausen, M. and A. Fortenbacher, "Efficient Solution of Linear Diophantine Equations". *J. Symbolic Computation 8*, 1989, pp. 201-216.

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.

Esparza, J. and M. Nielsen, "Decidability Issues for Petri Nets". Bulletin of the EATCS, Number 52, Feb. 1994.

Esparza, J., "Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes". *Proc. of Fundamentals of Computer Theory '95*, LNCS 965, 1995, pp. 221-232.

Fribourg, L. and M. Veloso Peixoto, "Bottom-up Evaluation of Datalog Programs with Incremental Arguments and Linear Arithmetic Constraints", *Proc. Post-ILPS'94 Workshop on Constraints and Databases*, Ithaca, N.Y., USA, 1994, pp. 109-125.

Fribourg, L. and H. Olsén, *Datalog Programs with Arithmetical Constraints: Hierarchic, Periodic an Spiralling Least Fixpoints.* Technical Report LIENS-95-26, Ecole Normale Supérieure, Paris, November 1995, 113 pages.

Fribourg, L. and H. Olsén, "Proving Safety Properties of Infinite State Systems by Compilation into Presburger Arithmetic". *Proc. 8th Intl. Conf. on Concurrency Theory*, Warsaw, Poland, LNCS, Springer-Verlag, 1997.

Van Gelder, A., "Deriving Constraints among Argument Sizes in Logic Programs". *Proc. 9th ACM Symp. on Principles of Database Systems*, Nashville, 1990, pp. 47-60.

Ginsburg, S. and E.H. Spanier, "Semigroups, Presburger formulas and languages". *Pacific Journal of Mathematics 16*, 1966, pp. 285-296.

Halbwachs, N., "Delay Analysis in Synchronous Programs". *Proc. Computer Aided Verification*, LNCS 697, Springer-Verlag, 1993, pp. 333-346.

Helm, A.R., "On the Detection and Elimination of Redundant Derivations during Bottom-up Execution". *Proc. North American Conference on Logic Programming*, Cleveland, Ohio, 1989, pp. 945-961.

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.

Jaffar, J. and J.L. Lassez, "Constraint Logic Programming". *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 111-119.

Kabanza, F., Stevenne, J.M. and P. Wolper, "Handling Infinite Temporal Data". *Proc. 9th ACM Symp. on Principles of Database Systems*, Nashville, 1990, pp. 392-403.

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).

Lassez, J.-L. and M.J. Maher, "The Denotational Semantics of Horn Clauses As a Production System". *Proc. AAAI-83*, Washington D.C., 1983, pp. 229-231.

Loveland, D.W., *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam, 1978.

Maher, M.J., "Equivalences of Logic Programs". *Foundations of Deductive Databases and of Logic Programming*, Ed. Jack Minker, Morgan Kaufmann Publishers, 1988, pp. 627-658.

Maher, M.J., "A Logic Programming View of CLP". *Proc. 10th Intl. Conf. on Logic Programming*, Budapest, 1993, pp. 737-753.

Melzer, S. and J. Esparza, "Checking System Properties via Integer Programming". SFB-Bericht 342/13/95A, Technische Universitaet Muenchen, September 1995. (See also: proceedings of ESOP '96)

Naughton, J.F. and Y. Sagiv, "A Decidable Class of Bounded Recursions". *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, 1987, pp. 171-180.

Nilsson, N.J., *Principles of Artificial Intelligence*, Springer Verlag, 1982.

Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc.,1981.

Plümer, L., "Termination Proofs for Logic Programs based on Predicate Inequalities". *Proc. 7th Intl. Conf. on Logic Programming*, Jerusalem, 1990, pp. 634-648.

Ramakrishnan, R., Sagiv, Y., Ulmann, J.D. and M.Y. Vardi, "Proof-Tree Transformation Theorems and their Applications". *Proc. 8th ACM Symp. on Principles of Database Systems*, Philadelphia, 1989, pp. 172-181.

Revesz, P., "A Closed Form for Datalog Queries with Integer Order". *Proc. 3rd International Conference on Database Theory*, Paris, 1990, pp. 187-201.

Srivastava, D. and R. Ramakrishnan, "Pushing Constraints Selections". *Proc. 11th ACM Symp. on Principles of Database Systems*, San Diego, 1992, pp. 301-315.

Tamaki, H. and T. Sato, "Unfold/fold Transformations of Logic Programs". *Proc. 2nd Intl. Conf. on Logic Programming*, Uppsala, 1984, pp. 127-138.

Verschaetse, K. and D. De Schreye, "Deriving Termination Proofs for Logic Programs using Abstract Procedures". *Proc. 8th Intl. Conf. on Logic Programming*, Paris, 1991, pp. 301-315.

Vieille, L., "Recursive Query Processing: The Power of Logic". *Theoretical Computer Science 69*, 1989, pp. 1-53.

Yen, H.-C., "On the Regularity of Petri Net Languages". *Information and Computation 124*, 1996, pp. 168-181.