A Decompositional Approach for Computing Least Fixed-points of Datalog Programs with Z-counters

Laurent FRIBOURG
Hans OLSÉN

LIENS - 96 - 12

A Decompositional Approach for Computing Least Fixed-points of Datalog Programs with Z-counters

Laurent FRIBOURG
Hans OLSÉN[*]

LIENS - 96 - 12

July 1996

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

Tel : (33)(1) 44 32 00 00
Adresse électronique : fribourg @dmi.ens.fr

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

Adresse électronique : hanol@ida.liu.se

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

Laurent Fribourg Hans Olsén

July 1996

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 together some unconnected results and extending them in several directions.

Résumé

Nous présentons une méthode pour caractériser les plus petits points-fixes d'une certaine classe de programmes Datalog dans l'arithmétique de Presburger. La méthode consiste à appliquer un ensemble de règles qui transforment les chemins généraux de calcul en chemins "canoniques". Nous utilisons cette méthode pour traiter le problème d'accessibilité dans le domaine des réseaux de Petri. Nous rapprochons ainsi des résultats considérés jusqu'ici comme indépendendants, et nous les étendons dans plusieurs directions.

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

Laurent Fribourg Hans Olsén

July 1996

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 together some unconnected results and extending them in several directions.

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 wether or not array bounds were violated. A pionneering work in this field is the work by Cousot-Halbwachs [10]. 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[16, 24, 28], compilation of recursive queries in temporal databases [2, 20], verification of safety properties of concurrent systems[18]. 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 [25] and of Chomicki-Imielinski [6] 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 [3], 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 by Esparza [13].

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 relates our results with those of Berthelot. Section 6 shows that our method allows us to solve the reachability problem for the special class of Basic Parallel Process nets. Section 7 interprets unfolding of propositional logic programs as a transformation of BPP nets. Section 8 gives a further generalized form of our basic decomposition rule. Section 9 briefly discusses the compilation into an arithmetic formula and our implementation. We conclude in section 9.

2 Preliminaries

Our aim in this report is to express the least model, of a certain class of logic programs, as a closed form logical formula over the domain $\mathcal{Z}$, [19][21]. We consider programs of the form:

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

for some constants $a_{i,l}, k_{i,j} \in \mathcal{Z}$. where $B(x_1,\ldots,x_m)$ is a linear integer relation (relation defined by a Presburger formula). This is more conveniently expressed as $\vartheta_{r_i}(\overline{x}) \equiv \overline{x} > \overline{a}_{r_i}$ for $\overline{a}_{r_i} \in (\mathcal{Z} \cup \{-\infty\})^m$, where, as usual, $n > -\infty$ for any integer $n \neq -\infty$, and where $-\infty \geq -\infty$ and $-\infty \pm n = n \pm (-\infty) = -\infty$ for any integer $n \neq -\infty$ holds. For any vectors $\overline{x}_1$ and $\overline{x}_2$, we define $\overline{x}_1 > \overline{x}_2$ (resp. $\overline{x}_1 \geq \overline{x}_2$) to hold, iff and only if the inequalities holds componentwise. $\max(\overline{x}_1, \overline{x}_2)$ is the vector obtained by taking the maximum of $\overline{x}_1$ and $\overline{x}_2$ componentwise. (thus, $\max(\overline{x}_1, \overline{x}_2)$ is the least upper bound in the $\langle(\mathcal{Z} \cup \{-\infty\})^m, \geq\rangle$-lattice. The vector with all components $-\infty$ is the bottom element.). The program is then written in a more appealing fashion, as:

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

Since $n > -\infty$ holds for any $n \in \mathcal{Z}$, any constraint of the form $x_{i,j} > -\infty$, is simply considered as *true*.

We want to express the least fixed-point as a linear integer arithmetic expression (a Presburger formula).

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 reason for considering the more general situation is partly because it can be done, and partly to gain insight into what is needed for our approach to work.

We introduce a convenient description of the execution, in a bottom-up manner, of programs of the general form above:

A *clause*, r, is a pair $\langle \overline{k}_r, \vartheta_r \rangle$, where $\vartheta_r \subseteq \mathcal{D}$ is a constraint (or "guard"). A clause r is *applicable* at a point $\overline{x} \in \mathcal{Z}^m$ iff $\vartheta_r(\overline{x})$ holds. The result of applying the rule r at a point $\overline{x}$ is $\overline{x}r = \overline{x} + \overline{k}_r$. More generally, let $\Sigma = \{r_1, \ldots, r_n\}$. A sequence $w \in \Sigma^*$ is called a *path*, and is interpreted as a sequence of applications of the clauses from right-to-left (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:

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

The expression $\vartheta_w(\overline{x})$ is said to be the *constraint* associated to path w at point $\overline{x}$.

By the definitions above, we get:

$$\begin{array}{ll} & \vartheta_{r_j w}(\overline{x}) \\ \Leftrightarrow & \vartheta_{r_i}(\overline{x}) \ \wedge \ \vartheta_w(\overline{x}r_i) \\ \Leftrightarrow & \overline{x} > \overline{a}_{r_i} \ \wedge \ \overline{x} + \overline{k}_{r_i} > \overline{a}_w \\ \Leftrightarrow & \overline{x} > \max(\overline{a}_{r_i}, \overline{a}_w - \overline{k}_{r_i}) \end{array}$$

where $\overline{a}_w - \overline{k}_{r_i}$ is well definied since $\overline{k}_{r_i} \in \mathcal{Z}^m$. It is immediately seen that, for programs with $\mathcal{Z}$-counters, the constraint associated with a path, is of the same form as that of a clause of the original program. In general, with every path w, there is associated a clause $\langle w \rangle = \langle \overline{k}_w, \vartheta_w \rangle$. And so, the class of programs with $\mathcal{Z}$-counters, is closed under the operation of considering the clauses associated with paths. That is, if Σ is a program with $\mathcal{Z}$-counters, then any *finite* language $L \subseteq \Sigma^*$, is also a program with $\mathcal{Z}$-counters. For instance, consider the two clauses (borrowed from an example given later on):

$$\begin{array}{lll} r_3: & p(x_2+1, x_3-1, x_4, x_5+1, x_6, x_7) & \leftarrow \ x_3 > 0, \ p(x_1, \ldots, x_7). \\ r_5: & p(x_2, x_3, x_4, x_5-1, x_6+1, x_7) & \leftarrow \ x_5 > 0, \ p(x_1, \ldots, x_7). \end{array}$$

The constraint $\vartheta_{r_3r_5}(\overline{x})$ associated with r_3r_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.: $x_3 > 0 \wedge x_5 > -1$, and the associated clause of the program is:

$$r_3r_5 : \qquad p(x_2+1, x_3-1, x_4, x_5, x_6+1, x_7) \ \leftarrow \ x_3 > 0,\ x_5 > -1,\ p(x_1, \ldots, x_7).$$

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. $\overline{x} \xrightarrow{\emptyset} \overline{x}' \ \Leftrightarrow\ false$
2. $\overline{x} \xrightarrow{\emptyset^*} \overline{x}' \ \Leftrightarrow\ \overline{x} \xrightarrow{\varepsilon} \overline{x}' \ \Leftrightarrow\ \overline{x} = \overline{x}'$
3. $\overline{x} \xrightarrow{w} \overline{x}' \ \Rightarrow\ \vartheta_w(\overline{x})$
4. $(L_1 \subseteq L_2) \ \Rightarrow\ \left(\overline{x} \xrightarrow{L_1} \overline{x}' \ \Rightarrow\ \overline{x} \xrightarrow{L_2} \overline{x}'\right)$
5. $\overline{x} \xrightarrow{L_1+L_2} \overline{x}' \ \Leftrightarrow\ \overline{x} \xrightarrow{L_1} \overline{x}' \vee \overline{x} \xrightarrow{L_2} \overline{x}'$
6. $\overline{x} \xrightarrow{L_1L_2} \overline{x}' \ \Leftrightarrow\ \exists\, \overline{x}'' :\ \overline{x} \xrightarrow{L_1} \overline{x}'' \xrightarrow{L_2} \overline{x}'$
7. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow\ \exists\ n \geq 0 :\ \overline{x}' = \overline{x} + n \cdot \overline{k}_w \ \wedge\ \forall\ 0 \leq n' < n :\ \vartheta_w(\overline{x} + n' \cdot \overline{k}_w)$

◇

The four last implications are the most important ones and, as will soon be explained, our method is based on these. Note, in the last implication, that if $n = 0$, then $\overline{x} = \overline{x}'$ and $\forall\ 0 \leq n' < n :\ \vartheta_w(\overline{x} + n' \cdot \overline{k}_w)$ is vacuously true. Also note that the last implication may be expressed as

7. $\overline{x} \xrightarrow{w^*} \overline{x}' \ \Leftrightarrow\ \exists\ n \geq 0 :\ \overline{x}' = \overline{x} + n \cdot \overline{k}_w \ \wedge\ \forall\ 0 \leq n' < n :\ \overline{x} + n' \cdot \overline{k}_w > \overline{a}_w$

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 $\overline{k}_w$ is $\langle 2, -3 \rangle$, then $\forall\ 0 \leq n' < n :\ \overline{x} + n' \cdot \overline{k}_w > \overline{a}_w$ is:

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

Therefore, the formula of (7) is equivalent to:

7. $\overline{x} \xrightarrow{w^*} \overline{x}' \Leftrightarrow \overline{x}' = \overline{x} \lor \exists\, n > 0 : \overline{x}' = \overline{x} + n \cdot \overline{k}_w \land \overline{x} + n \cdot \overline{k}_w^- > \overline{a}_w + \overline{k}_w^-$

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, define a *flat* language as:

1. Any finite language is flat.
2. w^* is flat for any $w \in \Sigma^*$.
3. $L_1 + L_2$ and $L_1 L_2$ are flat if L_1 and L_2 are flat.

We call such a language L "flat" because the Kleene's star operator '*' applies only to strings w. By proposition1, it follows that the relation $\overline{x} \xrightarrow{L} \overline{x}'$ for a flat language L, can be expressed as closed logical formula, and for programs with $\mathcal{Z}$-counters, in particular, as an existentially quantified formula of Presburger arithmetic, having $\overline{x}$ and $\overline{x}'$ as free variables. More precisely, the reachability is expressed as a disjunction of a number of matrix expressions of the form:

$$\exists\, \overline{n}_i : \overline{x}' = \overline{x} + K_i \overline{n}_i \land \overline{x} + C_i \overline{n}_i > \overline{a}_i$$

where K_i and C_i are matrices, and $\overline{a}_i$ some vector of constants.

Given a program with $B(\overline{x})$ as a base case and recursive clauses Σ, the least fixed-point of its immediate consequence operator (see [19][21]), which is also the least $\mathcal{Z}$-model of the program, may be expressed as:

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

Our aim is to characterize the membership relation $\overline{y} \in \text{lfp}$ as a closed formula having $\overline{y}$ as a free variable, and in particular, for programs with $\mathcal{Z}$-counters, as a linear arithmetic formula. For solving this problem, it suffices actually to characterize the relation $\overline{x} \xrightarrow{\Sigma^*} \overline{x}'$ as a closed 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 replaced by (usually reordered into) 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. In order to illustrate more concretely, let us already here state (without proof) the simplest and most obvious of the decomposition rules, called *stratification.*

Proposition 2:
Let $R, R' \subset \Sigma$ be sets of clauses such that $\Sigma = R \cup R'$, and such that

$$\overline{x} \xrightarrow{R'R} \overline{x}' \Rightarrow \overline{x} \xrightarrow{RR'} \overline{x}'$$

Then we have:

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

$\diamond$

Proposition 6 simply says, that any reachable point can be reached by a path such that every occurence of a clause belonging to R is before all occurrences of clauses of R'. Or stated differently, any path $w_1 r' r w_2$, can always be replaced by a path $w_1, tt' w_2$, where $r, t \in R$ and $r', t' \in R'$. By repeating this replacement, one must sooner or later get a path belonging to $R^* R'^*$. The point is that the precondition of the decomposition involves only *finite* languages (viz, $R'R$ and RR') and can therefore be checked, while the conclusion refers to infinite languages (viz, Σ^* and $R^* R'^*$). By simply going back to the definition of $\overline{x} \xrightarrow{L} \overline{x}'$, one sees that the precondition reduces to verifying a number of inequalities among constants:

$$\bigwedge_{r'r \in R'R} \quad \bigvee_{tt' \in RR'} \max(\overline{a}_{r'}, \overline{a}_r - \overline{k}_{r'}) \geq \max(\overline{a}_t, \overline{a}_{t'} - \overline{k}_t)$$

An obvious sufficient condition (the most common situation) is:

$$\bigwedge_{r'r \in R'R} \max(\overline{a}_{r'}, \overline{a}_r - \overline{k}_{r'}) \geq \max(\overline{a}_r, \overline{a}_{r'} - \overline{k}_r)$$

That is, the rules commute: $\forall r \in R, r' \in R' : \overline{x} \xrightarrow{r'r} \overline{x}' \Rightarrow \overline{x} \xrightarrow{rr'} \overline{x}'$. Consider for example the program:

$$\begin{array}{lllll}
 & p(x_1, x_2) & \leftarrow & x_1 = 5, x_2 = -13. \\
r_1 : & p(x_1 - 1, x_2 + 3) & \leftarrow & x_1 > 1, \ p(x_1, x_2). \\
r_2 : & p(x_1 - 2, x_2 - 1) & \leftarrow & x_2 > -3, \ p(x_1, x_2).
\end{array}$$

In proposition 6, choose, $R = \{r_1\}$ and $R' = \{r_2\}$. To check the precondition, consider:

$$\begin{array}{ll}
 & \overline{x} \xrightarrow{R'R} \overline{x}' \Rightarrow \overline{x} \xrightarrow{RR'} \overline{x}' \\
\Leftrightarrow & \overline{x} \xrightarrow{r_2 r_1} \overline{x}' \Rightarrow \overline{x} \xrightarrow{r_1 r_2} \overline{x}' \\
\Leftrightarrow & \max(\langle -\infty, -3\rangle, \langle 1, -\infty\rangle - \langle -2, -1\rangle) \geq \max(\langle 1, -\infty\rangle, \langle -\infty, -3\rangle - \langle -1, 3\rangle) \\
\Leftrightarrow & \max(\langle -\infty, -3\rangle, \langle 3, -\infty\rangle) \geq \max(\langle 1, -\infty\rangle, \langle -\infty, -6\rangle) \\
\Leftrightarrow & \langle 3, -3\rangle \geq \langle 1, -6\rangle
\end{array}$$

which is true, and therefore, by proposition 6, the fixpoint is given by

$$\text{lfp} = \{ \ \langle x_1', x_2'\rangle \mid \langle 5, -13\rangle \xrightarrow{r_1^* r_2^*} \langle x_1', x_2'\rangle \ \}$$

Since $r_1^* r_2^*$ is a flat language, by proposition 1.6 and 1.7'', we get (after arithmetical simplification):

$$\begin{array}{lll}
p(x_1, x_2) & \Leftrightarrow & (3x_1 + x_2 - 2 = 0 \ \wedge \ 5 \geq x_1 \geq 0) \ \vee \\
 & & (35 \geq x_1 - 2x_2 + 4 \geq 0 \ \wedge \ 0 > x_2 > 4)
\end{array}$$

Thus, the preconditions of the decomposition rules are essentially syntactic criteria on the program, and the proof of correctness of the decomposition rules, becomes a combinatorial problem of showing how any path in general can be reordered under various assumptions of "replaceability".

Such a "decompositional approach" is well known in Petri-net theory (see, e.g., [29]). The rest of the report 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.

Since we will reason a lot about languages (mainly regular expressions), we introduce some convenient notation. Although we have already used concatenation, sum and Kleene closure, we give their definitions for the sake of completeness. We will also introduce some special notation that do not extend the class of regular languages, but provide us with some short hands.

Let us simply list the definitions:

1. $L_1 + L_2 = L_1 \cup L_2$. Thus, by abuse of notation we consider $+$ and $\cup$ as synonymous.
2. $L_1 L_2 = \{\ vw \mid v \in L_1 \text{ and } w \in L_2\ \}$
3. $L^0 = \{\varepsilon\}$ and $L^{n+1} = LL^n$. It is practical to extend the definition of exponentiation to allow a negative exponent as follows: $L^{-n} = \emptyset$ when $n > 0$.
4. $L^{\leq k} = \bigcup_{n \leq k} L^n$. By 3, if $k < 0$, then $L^{\leq k} = \emptyset$.
5. $L^{<k} = \bigcup_{n<k} L^n$. By 3, we have $L^{<0} = \emptyset$.
6. $L^{k_1 < * < k_2} = \bigcup_{k_1 < n < k_2} L^n$. Here, if $k_1 = k_2$, then $L^{k_1 < * < k_2} = \emptyset$ since in that case the union is empty.
7. For the sake of completeness: $L^* = \bigcup_{n=0}^{\infty} L^n$, and $L^+ = \bigcup_{n=1}^{\infty} L^n$.

Let us here make an observation useful for proofs by induction. For any language L, we have:

$$\overline{x} \xrightarrow{L^*} \overline{x}' \Leftrightarrow \exists n \geq 0 : \overline{x} \xrightarrow{L^n} \overline{x}'$$

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 [12].) 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 $P \cup P'$: $p(\overline{x}) \ \Leftrightarrow\ \exists x'_j\ p'(\overline{x}, x'_j)$.

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 1:

We consider here a Petri net implementing a simple readers-writers protocol. (This is inspired from [1], 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

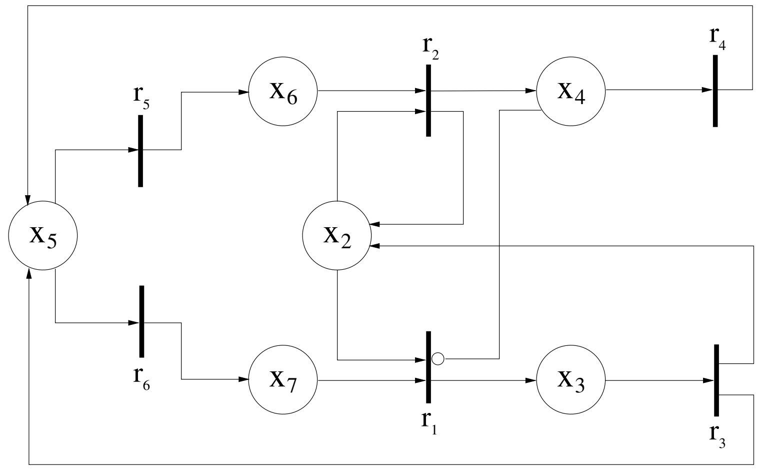


Figure 1

this Petri-net is the following:

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

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:

$$p'(x_1,x_2,x_3,x_4,x_5,x_6,x_7) \leftarrow x_1 = 1 - x_4, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = q \geq 0, x_6 = 0, x_7 = 0.$$

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

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., [5]) are able to prove the first implication: by analysing the transitions without taking into account the guards, they generate a set of invariants among which 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 [22] 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 2:
The program P' of example 1 is represented by:

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

$\diamond$

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 only a few transformation rules for lack of space. The rules are stated on the form: $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{L_1 L_2 \cdots L_s} \overline{x}'$. 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 *invariant* variables of the program (i.e. the number of 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 4.2, 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 because no decomposition rule can be applied.

In keeping with a former approach [14], 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.

4.1.1 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 has "priority" over 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 3:
Let $r_l \in \Sigma$ be a clause such that $\forall j : k_{l,j} \geq 0$. Then:

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

$\diamond$

Proof
Since, for all $r_j \in \Sigma$, the coefficients $k_{l,j}$ is nonnegative, the constraint ϑ_{r_j} is invariant under the application of r_l (i.e.: $\vartheta_{r_j}(\overline{x}) \Rightarrow \vartheta_{r_j}(\overline{x}r_l)$). 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 **C**-equivalence, 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.

4.2 cyclic decomposition rule

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

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

where R and R' are sets of rules such that $\Sigma = R \uplus R'$, the constraints of all the clauses in R are exactly $x_j > 0$ and x_j does not occur in the constraints of any rule in R'. Formally this means

1. $\forall r_i \in R : \; k_{i,h} \geq 0$ for $h \neq j$

1'. $\forall r_i \in R' : \; x_j$ does not occur in $\vartheta_{r_i}(\overline{x})$

2. $\forall r_i \in R' : \; k_{i,j} \geq 0$

3. $\forall r_i \in R : \; k_{i,j} = -1$

4. $\forall r_i \in R : \; \vartheta_{r_i}(\overline{x}) \equiv x_j > 0$

Under conditions $1, 1', 2, 3, 4$, given a path w starting at a point where x_j is greater than 0, one can reorder w so that all the R-clauses are applied first (similarly to the situation of the monotonic transformation), but now such a priority of application for the R-clauses must end at some point: this is because, here, the coefficients $k_{i,j}$ $(l+1 \leq i \leq n)$ 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: $\quad x_j > 0 \ \wedge \ \overline{x} \xrightarrow{R'R} \overline{x}' \ \Rightarrow \ \overline{x} \xrightarrow{RR'} \overline{x}'$.

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:

$p(\overline{x} + \overline{k}_{r_i w}) \leftarrow \vartheta_{r_i w}(\overline{x}), p(\overline{x}),\quad$ 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 4:

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

$$
\begin{array}{l}
x_j \geq 0 \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \;\Rightarrow \\
\qquad \overline{x} \xrightarrow{R^* R'^*} \overline{x}' \\
\qquad\qquad \vee \\
\qquad \exists \overline{x}'' : \; \overline{x} \xrightarrow{R^*} \overline{x}'' \xrightarrow{\Sigma^*} \overline{x}' \wedge x''_j = 0
\end{array}
$$

B

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

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

C

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

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

$\diamond$

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 less clauses than Σ^* (and at least as many variables let 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 lets 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, $\overline{x} \xrightarrow{v'} \overline{x}' \vee \exists \overline{x}''' : \overline{x}'' \xrightarrow{v''} \overline{x}''' \xrightarrow{\Sigma^*} \overline{x}' \wedge x''_j = 0$, 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 $|v| < n$, $x_j = 0 \wedge \overline{x} \xrightarrow{v} \overline{x}' \Rightarrow \overline{x} \xrightarrow{LR'^*} \overline{x}'$, 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 $\overline{x} \xrightarrow{r_i LR'^*} \overline{x}'$. This proves $\overline{x} \xrightarrow{LR'^*} \overline{x}'$, since $r_i L \subseteq L$. Therefore assume $k_{i,j} > 0$, in wich 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 $\overline{x} \xrightarrow{r_i w''_1 LR'^*} \overline{x}'$, which proves $\overline{x} \xrightarrow{LR'^*} \overline{x}'$, since $r_i v'' L \subseteq L$. 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 $r_i u w_1'' \in \left(\varepsilon + \bigcup_{r_i \in R'} r_i R^{0<*<k_{i,j}}\right) R'^* \subseteq LR'^*$, 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.)

Example 3:
Consider the matrix in example 2, representing the program for the protocol of example 1. Let us in proposition 4 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 4 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 4.C, we have:

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

$$\begin{aligned}
&\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}' \\
&\qquad\qquad \vee \\
&\exists \overline{x}'' : \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*} \overline{x}'' \wedge \\
&\qquad \overline{x}'' \xrightarrow{(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}' \wedge x_5'' = 0
\end{aligned}$$

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.)

$\diamond$

5 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 the proposition, to a point of coordinate $x_5 = 0$ and x_5 is let invariant. So an obvious optimization, for the treatment of the matrix, will be to remove the null column as well as the guard $x_5 > -1$. In terms of Petri nets, this corresponds to remove the place x_5 and to perform the "fusion" of transitions r_2, r_3, r_4 (which have x_5 as an output place) and transitions r_5, r_6 (which have x_5 as an input place). The resulting Petri net is represented in figure 2. This kind of

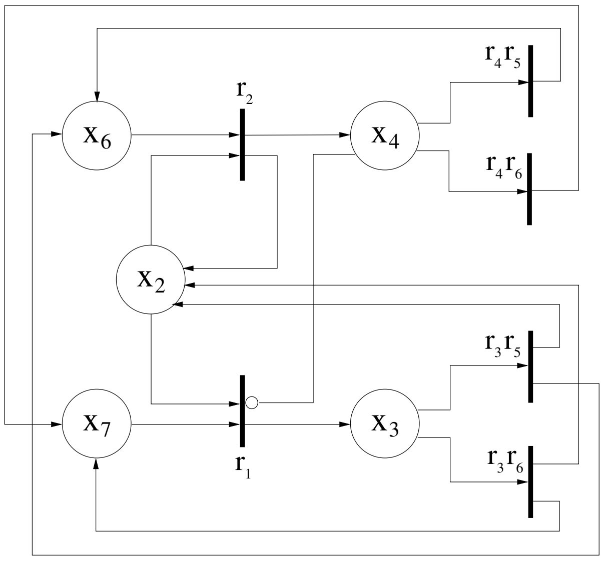


Figure 2

optimization can be done generally, under the preconditions of proposition 3. An analogous transformation of Petri nets is called *post-fusion* transformation by Berthelot in [3]. Our version of the cyclic decomposition can thus be seen as a variant of Berthelot's post-fusion rule. Berthelot also defined in [3] some other transformations like *pre-fusion*. It is possible to give in our framework a counterpart also for this transformation, although there is no place here to present it. 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 a next section 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.)

6 Application to BPP-nets

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 [1] one input place and removes exactly one token from that one place. BPP stands for Basic Parallel Process: this is a class of CSS process defined by Christensen [7]; the reachability problem for BPP-nets is NP-complete [13]. 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 $x_h > 0$. Let us show that a sequence of the two decomposition rules presented above is always applicable to such a program (so that, eventually, we are sure to get a flat language). If there exists a clause r_i such that all the coefficients of its head are nonnegative, the monotonic (increasing) decomposition rule is applied, and the problem is decomposed into $(\Sigma - \{r_i\})^* r_i^* (\Sigma - \{r_i\})^*$. If there are still such clauses in $\Sigma - \{r_i\}$, we apply the monotonic rule until every clause contains a negative coefficient. By assumption, every clause must then contain *exactly one* negative coefficient and it must be equal to -1. Let us show that the cyclic decomposition rule then applies. 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 4. 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 $\Sigma - R$. Thus, for every $r_i \in R'$ we have $k_{i,j} \geq 0$, and therefore conditions 1 to 4 of the specialized case on page 11 are satisfied, so the cyclic decomposition applies. It is easy to see that the new programs generated are still of the form corresponding to a BPP-net, so one of the two decompositions (monotonic clause or cyclic "post fusion" rule) is always applicable. Thus, for this class of programs 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.

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

This yields a new proof of the fact that the reachability set for BPP-nets is a semi linear set [13]. 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 [17]) 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 $\overline{a}^0$).

7 Propositional Logic Programs and BPP Nets

We show here how to simulate linear resolution via a propositional logic program π by firing transitions of an associated BPP-net. We also explain how to simulate unfolding of π by fusing transitions of the BPP-net.

7.1 Linear resolution of clauses as firing of transitions

Consider the following propositional logic program π

$\gamma_1 : \ f \leftarrow a, b$

$\gamma_2 : \ a \leftarrow b, c$

$\gamma_3 : \ a \leftarrow d$

Suppose that you want to solve a goal of the form $\leftarrow b, f, b$. Then you are going to match a literal, say f, of the goal with the head of a program clause, viz. γ_1, and replace it by the clause body, thus yielding the new goal $\leftarrow b, a, b, b$. This corresponds to a step of *linear resolution*. The process can be iterated and ends when you get the empty goal $\leftarrow$, which means that you have obtained a *linear refutation* of the initial goal $\leftarrow f$. This can be also interpreted as a *proof by backward chaining* of the positive literal f.

Let us show that one can encode the propositional program logic π under the form of a BPP-net, and mimick linear clausal resolution by firing net transitions. The BPP-net is constructed by associating a place with each literal (of the Herbrand base) of the program, and associating a transition with each clause. For example, the net in figure 3 will be associated with clauses $\gamma_1, \gamma_2, \gamma_3$. Transition r_1 (resp. r_2, r_3) corresponds to clause γ_1 (resp. γ_2, γ_3). Places x_a, x_b, x_c, x_d, x_f correspond to literals a, b, c, d, f respectively. The head of clause γ_i is represented as an input place of the corresponding transition r_i while the literals of the body are represented as output places of r_i. The net is a BPP-net (only one input place by transition) because the clauses of the original program are definite Horn clause (with a single literal by head). (Note the "structure sharing" among the places of the net; there is no duplication of place: for example, place x_b corresponding to literal b, which appears twice in clauses γ_1 and γ_2, is shared as an input place of both transitions r_1 and r_2.)

The reachability predicate p associated with this net is defined by the following program Σ_π:

$\rho_1 : \ p(x_f - 1, x_a + 1, x_b + 1, x_c, x_d) \leftarrow p(x_f, x_a, x_b, x_c, x_d), x_f > 0.$

$\rho_2 : \ p(x_f, x_a - 1, x_b + 1, x_c + 1, x_d) \leftarrow p(x_f, x_a, x_b, x_c, x_d), x_a > 0.$

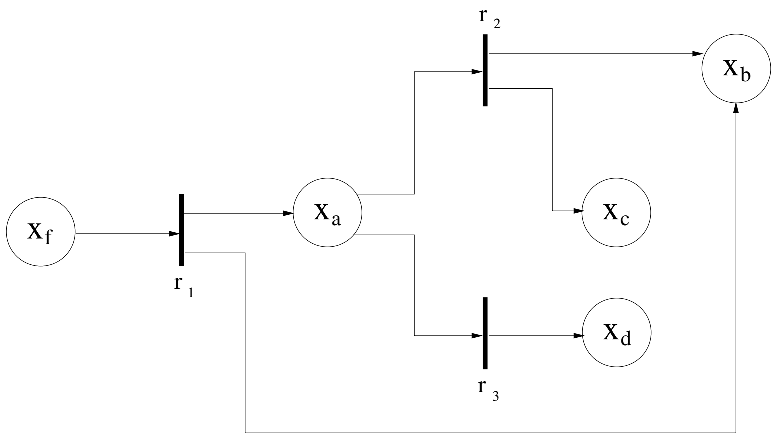


Figure 3

$\rho_3: \ p(x_f, x_a - 1, x_b, x_c, x_d + 1) \leftarrow p(x_f, x_a, x_b, x_c, x_d), x_a > 0.$

, which is more concisely represented by the matrix:

$$\begin{array}{rrrrrccl}
-1 & 1 & 1 & 0 & 0 & & x_f > 0 & : \rho_1 \\
0 & -1 & 1 & 1 & 0 & & x_a > 0 & : \rho_2 \\
0 & -1 & 0 & 0 & 1 & & x_a > 0 & : \rho_3 \\
x_f & x_a & x_b & x_c & x_d & & &
\end{array}$$

In this context a marking should be interpreted as a *conjunction of literals*. For example the marking $\langle 1,0,2,0,0\rangle$ means $f \wedge b \wedge b$. With this interpretation in mind, it is easy to simulate linear resolution. For example consider the resolution of goal $\leftarrow b, f, b$ via clause γ_1, which gives the new goal $\leftarrow b, a, b, b$. This is simulated by starting with marking $\langle 1,0,2,0,0\rangle$ on the above BPP-net, and firing transition r_1, which gives marking $\langle 0,1,3,0,0\rangle$. Goals thus correspond to markings, and resolution via program clauses to firing of net transitions. Refuting a goal g via linear resolution corresponds to starting from the marking associated with g, and reaching the empty marking through a certain sequence of fired transitions.

Given a marking $\overline{x}_0$, the atom $p(\overline{x}_0)$, should be interpreted as: "goal $\leftarrow \overline{x}_0$ is refuted", or in a positive way as: "conjunction $\overline{x}_0$ is proved". Program Σ_π thus encodes for the *provability* via π. Each clause ρ_i of program Σ_π encodes for *backward inference* via the corresponding clause γ_i of program π.

7.2 Unfolding of clauses as postfusion of transitions

Let us now recall the idea of *unfolding* (we paraphrase here [26], p.147).
Suppose that we use clause $\gamma_1: \ f \leftarrow a, b$ to replace f by $a \wedge b$ within a goal. We can only get an answer to the goal by subsequently eliminating a. This must be done using one of the clauses

$\gamma_2: \ a \leftarrow b, c$
$\gamma_3: \ a \leftarrow d$

so the same result could be achieved in one step using one of the unfolded clauses

$\gamma_1' : \ f \leftarrow b, c, b$
$\gamma_2' : \ f \leftarrow d, b$
So, the effect of unfolding is to short circuit (and hence shorten) the derivation.
We are going to show that such an unfolding step on propositional Horn clauses can be simulated by a post-fusion step on the program Σ_π associated with program π.
Consider the program Σ_π associated with program $\pi : \{\gamma_1, \gamma_2, \gamma_3\}$. Its incrementation matrix is:

$$\begin{array}{ccccccc} -1 & 1 & 1 & 0 & 0 & x_f > 0 & : \rho_1 \\ 0 & -1 & 1 & 1 & 0 & x_a > 0 & : \rho_2 \\ 0 & -1 & 0 & 0 & 1 & x_a > 0 & : \rho_3 \\ x_f & x_a & x_b & x_c & x_d & & \end{array}$$

One can apply post-fusion on place x_a to this matrix. This gives:

$$\begin{array}{ccccccc} -1 & 0 & 2 & 1 & 0 & x_f > 0 & : \rho_1.\rho_2 \\ -1 & 0 & 1 & 0 & 1 & x_f > 0 & : \rho_1.\rho_3 \\ x_f & x_a & x_b & x_c & x_d & & \end{array}$$

The fused net can be represented as in figure 4.

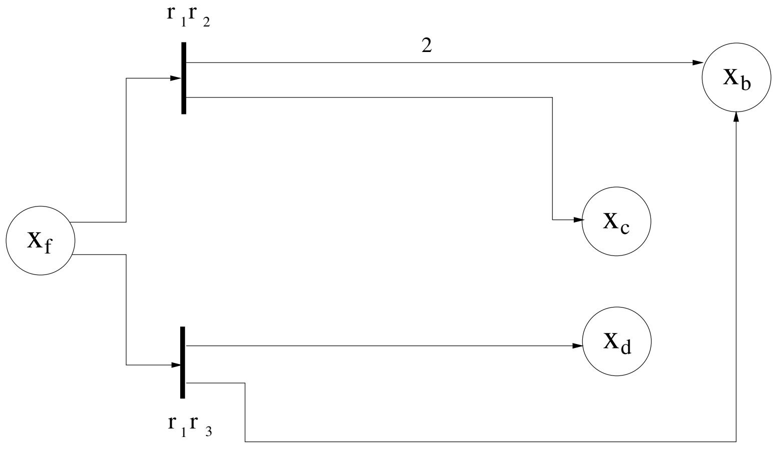


Figure 4

(Note that place x_b is shared here not only among distinct transitions, but also within a same transition by means of an input arc with weight equal to 2).

Clause $\rho_1\rho_2$ encodes for (backward inference via) a clause having f as a head, and b, b, c as a body, i.e.: $f \leftarrow b, b, c$. Clause $\rho_1\rho_3$ encodes for (backward inference via) a clause having f as a head and b, d as a body, i.e.: $f \leftarrow b, d$. One thus retrieves the two clauses γ_1' and γ_2' of the unfolded program.

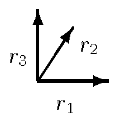
We have thus shown on this little example that post-fusion on the program Σ_π encoding provability via the propositional logic program π, simulates unfolding of π. More precisely, let a be some literal of π, and π' the result of unfolding clauses of π on to a. Let Σ' be the program obtained from Σ_π by postfusing Σ_π on to x_a, then Σ' coincides, up to the order of clauses and literals, with the program $\Sigma_{\pi'}$ encoding for provability via π'.

8 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 formulations 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 5, 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:



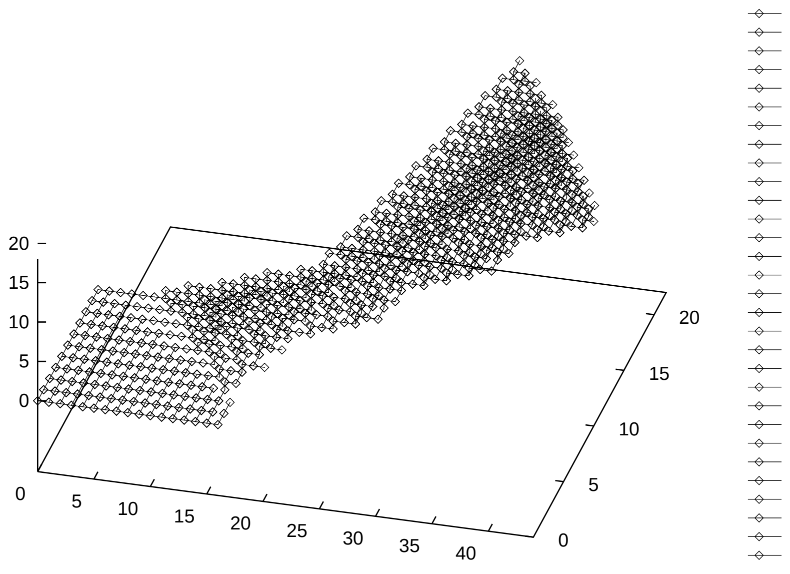


Figure 5

Let us choose x_j to be x_3, R' to be $\{r_1, r_2\}$ and R to be $\{r_3\}$. A priori, proposition 4 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 4 says, we have: $\langle 30,19,-57\rangle \xrightarrow{(r_1+r_2+r_3)^*} \overline{x}' \;\Rightarrow\; \langle 30,19,-57\rangle \xrightarrow{(r_1+r_2)^* r_3^* L (r_1+r_2)^*} \overline{x}'$ 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 5. 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 ot 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 4, 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). [2] 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 6. The construction is identical to that by Clausen and Fortenbacher [9] for solving

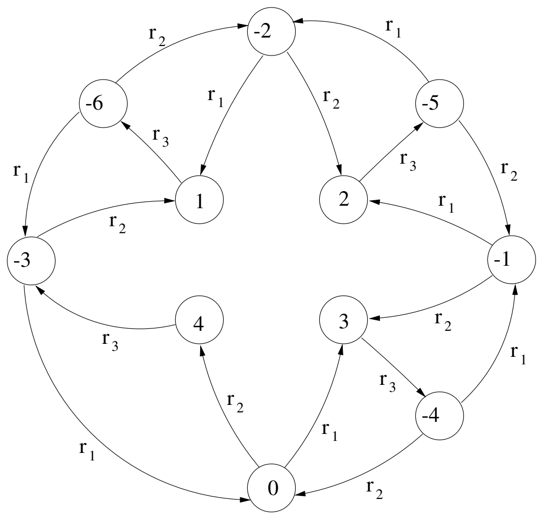


Figure 6

the linear diophantine equation:

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

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

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:

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

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 reach 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 5 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 4.

Proposition 5:
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

$$\begin{array}{l} x_j > c \wedge \overline{x} \xrightarrow{\Sigma^*} \overline{x}' \ \Rightarrow \\ \quad \overline{x} \xrightarrow{R^* R'^*} \overline{p}' \\ \qquad\qquad \vee \\ \quad \exists \overline{x}'' :\ x_j > c \wedge \overline{x} \xrightarrow{R^*} \overline{x}'' \xrightarrow{\Sigma^*} \overline{x}' \wedge b \geq x''_j > c \end{array}$$

B

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

C

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

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

◇

As can be seen on figure 6, 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 5 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:

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

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 [15]).

We can look back at the results stated in proposition 4, and interpret them as a special case of the above automaton-based construction. Under the conditions of proposition 4, 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 4, can now be constructed, using this specialized automaton, as illustrated on figure 7. Figure 8 gives a geometrical inter-

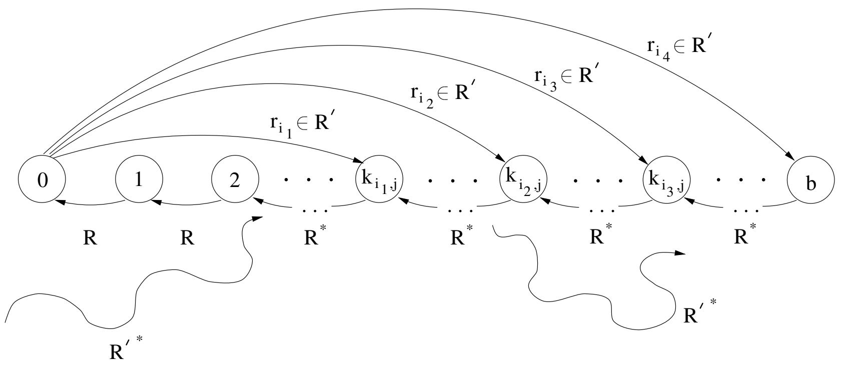


Figure 7

pretation of the fact all the cycles closely follow the hyper plane $x_j = 0$.

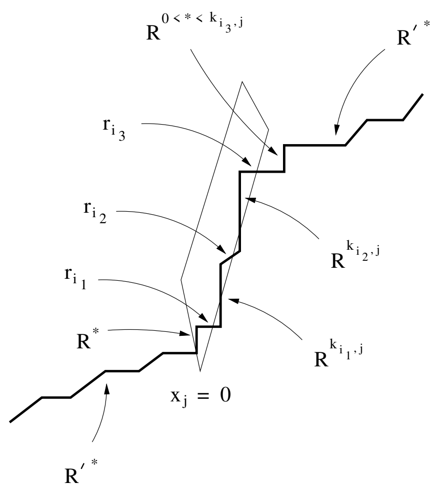


Figure 8

9 Compilation into Arithmetic

We have an experimental implementation in SICSTUS-PROLOG currently containing nine decomposition rules, four of which are cyclic. At the moment the implementation does not include any optimization for avoiding unnecessary decompositions, and no arithmetic is done. The program simply decomposes the reachability relation and outputs a regular expression that could be compiled into a linear arithmetic formula. For the readers-writers protocol of figure 1, our program successfully constructs a flat language $L \subseteq \Sigma^*$ such that $\overline{x} \xrightarrow{\Sigma^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{L} \overline{x}'$ and:

$$\begin{aligned} L \;=\; & r_5^*r_6^*r_2^*r_1^*r_2^*r_4^*(r_2r_4)^*r_5^*(r_2r_4)^*r_5^*r_6^*r_2^*r_1^*r_2^*r_3^*(r_1r_3)^*r_2^*r_4^*(r_1r_3)^*(r_2r_4)^*(r_1r_3)^*r_2^*r_5^*r_6^* \\ & (r_1r_3r_5)^*r_2^*(r_4r_6)^*(r_4r_5)^*(r_1r_3r_5)^*(r_2r_4r_6)^*(r_1r_3r_5)^*(r_1r_3)^*r_2^*r_4^*(r_1r_3)^*(r_2r_4)^* \\ & (r_1r_3)^*r_5^*r_6^*r_2^*r_1^*r_2^*r_4^*(r_2r_4)^*r_5^*(r_2r_4)^*r_5^*r_6^*r_2^*r_1^*r_2^* \end{aligned}$$

The expression consists of 51 factors and was computed in 15 seconds on a SPARC-10 machine. The decomposition was achieved by using 10 applications of the stratification rule (see appendix A), 7 applications of the cyclic "post-fusion" decomposition presented above, 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 fixpoint 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:

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

When compiling the arithmetic formula, at each step one checks wether the reachability set 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. In the present case of the 51 strings long expression, it can be shown that

the least fixed-point is reached after only 4 steps (see appendix B). So, the fixed-point of the program P' thus turned out to be given by

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

and out of the 51 factors in the language L, only the first 4 are needed. The arithmetical expression of this least fixed-point is given by:

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

It is immediately seen that the mutual exclusion property, $x_3 = 0 \ \vee\ x_4 = 0$, holds.

We are currently working on implementing linear integer constraint solving in order to fully automate the process. Our preliminary experiences with the reader-writers example as well as a couple of parametrized protocols [4][8] are very encouraging.

10 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 relating together 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 inhibitors and input arcs picking up more than one token). Our system implementation gives already promising results, as illustrated here on the readers-writers protocol. We plan to apply also our method in other fields such as the derivation of interargument relations for proving the termination of Prolog programs (see [11]) or the automatic generation of lemmas (see [14] and appendix D).

Acknowledgements. Special thanks are due to Alain Finkel for providing many useful informations on Petri nets.

References

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

[2] M. Baudinet, M. Niezette 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.

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

[4] A. Bourguet-Rouger. "External Behaviour Equivalence Between two Petri Nets". *Concurrency 88*, LNCS 335, Springer-Verlag, 1988, pp. 237-258.

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

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

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

[8] E.W. Dijkstra, W.H.J. Feijen and A.J.M. Van Gasteren. "Derivation of a termination detection algorithm for distributed computations", *Information Processing Letters 16:5*, 1983, pp. 217-219.

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

[10] P. Cousot 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.

[11] D. De Schreye and S. Decorte. "Termination of Logic Programs: The Never-Ending Story", *J. Logic Programming 19-20*, 1994, pp. 199-260.

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

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

[14] L. Fribourg 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.

[15] L. Fribourg 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.

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

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

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

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

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

[21] P. Kanellakis, G. Kuper and P. Revesz. "Constraint Query Languages". Internal Report, November 1990. (Short version in *Proc. 9th ACM Symp. on Principles of Database Systems*, Nashville, 1990, pp. 299-313).

[22] S. Melzer 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)

[23] J. Misra and D. Gires. "Finding Repeated Elements", *Science of Computer Programming 2*, 1992, pp. 143-152.

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

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

[26] J. C. Shepherdson, "Unfold/fold transformations of logic programs". *Math. Struct. in Comp. Science*, 1992, vol. 2, pp. 143-157

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

[28] K. Verschaetse 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.

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

APPENDIX A: two other monotonic decompositions

We present here two other decomposition rules sufficient for the full treatment of the readers-writers protocol of example 1. The first rule is called *stratification* and simply states that some clauses can be applied before all the others.

Proposition 6:
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" [27] and applies when there is a single-signed column. We present here only the decreasing version.

Proposition 7:
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. ◇

APPENDIX B

Let us compute the fixpoint of the program P' of example 1 from the language $L = r_5^* r_6^* r_2^* r_1^* \ldots$ of section 9, generated by our program. We get the sequence (making arithmetic simplifications at each step):

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

One may easily check that $\forall r_i \in \{r_1, \ldots, r_6\} : \ \vartheta_{r_i}(\overline{x}) \wedge \xi_4(\overline{x}) \ \Rightarrow \ \xi_4(\overline{x} r_i)$.

APPENDIX C

We give here a different decomposition (done by hand) of the readers writers protocol to illustrate some principles of optimization.

In example 3, by applying proposition 4, the reachability relation was decomposed as: $\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*(r_5+r_6)^*(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$. However, the ultimate goal is to compute an expression for $\text{lfp}_{P'} = \{\overline{x}' \mid \exists \overline{x} : B(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5+r_6)^*} \overline{x}'\}$ for the program P' of example 1, where:

$$\begin{array}{rcl} \xi_0(x_1,x_2,x_3,x_4,x_5,x_6,x_7) & \Leftrightarrow & B(x_1,x_2,x_3,x_4,x_5,x_6,x_7) \\ & \Leftrightarrow & x_1 = 1 - x_4 \ \wedge \ x_2 = 1 \ \wedge \ x_3 = 0 \ \wedge \\ & & x_4 = 0 \ \wedge \ x_5 = q \geq 0 \ \wedge \ x_6 = 0 \ \wedge \ x_7 = 0 \end{array}$$

Clearly $\xi_0(\overline{x}) \Rightarrow x_5 \geq 0$ holds. But then, by proposition 4.A combined with B, the least fixpoint is actually described by: $\xi_0(\overline{x}) \wedge \overline{x} \xrightarrow{(r_5+r_6)^*(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$ That is, the leftmost sublanguage, $(r_1+r_2+r_3+r_4)^*$, has been dropped. By taking the base clause of the program into account, the size of the linear arithmetic expression, ultimately computed, may be significantly reduced. It is easy to see that $\overline{x} \xrightarrow{r_6r_5} \overline{x}' \Rightarrow \overline{x} \xrightarrow{r_5r_6} \overline{x}'$, so by stratification (proposition 6), we have: $\overline{x} \xrightarrow{(r_5+r_6)^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{r_5^*r_6^*} \overline{x}'$. Since $r_5^*r_6^*$ is a simple language, we can apply proposition 1 and define (after some arithmetic simplifications):

$$\begin{array}{rll} \xi_1(\overline{x}') & \Leftrightarrow & \exists \overline{x} : \qquad \xi_0(\overline{x}) \ \wedge \ \overline{x} \xrightarrow{r_5^*r_6^*} \overline{x}' \\ & \Leftrightarrow & x'_1 = 1 \ \wedge \ x'_2 = 1 \ \wedge \ x'_3 = 0 \ \wedge \ x'_4 = 0 \ \wedge \ q \geq 0 \ \wedge \\ & & \exists n_5, n_6 \geq 0 : \ x'_5 = q - n_5 - n_6 \ \wedge \ x'_6 = n_5 \ \wedge \ x'_7 = n_6 \ \wedge \\ & & \qquad x'_5 + 1 > 0 \\ & \Leftrightarrow & x'_1 = 1 \ \wedge \ x'_2 = 1 \ \wedge \ x'_3 = 0 \ \wedge \ x'_4 = 0 \ \wedge \\ & & x'_5 \geq 0 \ \wedge \ x'_6 \geq 0 \ \wedge \ x'_7 \geq 0 \ \wedge \ x'_5 + x'_6 + x'_7 = q \end{array}$$

Thus: $\text{lfp}_{P'} = \{\overline{x}' \mid \exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2+r_3r_5+r_3r_6+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'\}$. Consider the leftmost sublanguage and choose $R = \{r_3r_5, r_3r_6\}$. Since $B'(\overline{x}) \Rightarrow x_3 = 0$, by proposition 4.B, the expression: $\exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1r_3r_5+r_2+r_4r_5+r_4r_6)^*(r_1+r_2+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$, describes the reachability set, and the matrix associated with the leftmost sublanguage is:

$$\begin{array}{rrrrrrrll} 0 & 0 & 0 & 0 & 0 & 1 & -1 & x_2 > 0, x_7 > 0, x_1 > 0, x_3 > -1, x_5 > -1 & : r_1r_3r_5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_2 > 0, x_7 > 0, x_1 > 0, x_3 > -1, x_5 > -1 & : r_1r_3r_6 \\ -1 & 0 & 0 & 1 & 0 & -1 & 0 & x_2 > 0, x_6 > 0 & : r_2 \\ 1 & 0 & 0 & -1 & 0 & 1 & 0 & x_4 > 0, x_5 > -1 & : r_4r_5 \\ 1 & 0 & 0 & -1 & 0 & 0 & 1 & x_4 > 0, x_5 > -1 & : r_4r_6 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & & \end{array}$$

The clause $r_1r_3r_6$ has all coefficients equal to zero so it may be safely deleted. We also see that the second column happened to become zero. Next we choose $R = \{r_4r_5, r_4r_6\}$

and since $\xi_1(\overline{x}) \;\Rightarrow\; x_4 = 0$ we get, in the same way as above (after deleting the zero clause $r_2r_4r_5$):

$$\begin{array}{rrrrrrrll}
0 & 0 & 0 & 0 & 0 & 1 & -1 & x_2 > 0, x_7 > 0, x_1 > 0, x_3 > -1, x_5 > -1 & : r_1r_3r_5 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & x_2 > 0, x_6 > 0, x_4 > -1, x_5 > -1 & : r_2r_4r_6 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

where $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1r_3r_5+r_2r_4r_6)^*(r_1r_3r_5+r_2)^*(r_1+r_2+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$ describes the reachability set. Now note that $\vartheta_{r_1r_3r_5}(\overline{x}) \wedge \xi_1(\overline{x}) \;\Rightarrow\; \xi_1(\overline{x}r_1r_3r_5)$ and $\vartheta_{r_2r_4r_6}(\overline{x}) \wedge \xi_1(\overline{x}) \;\Rightarrow\; \xi_1(\overline{x}r_2r_4r_6)$ hold. This means that the clauses of $r_1r_3r_5$ and $r_2r_4r_6$ does not produce any points not allready in the set computed so far. Therefore the leftmost sublanguage is simply dropped, which saves us the effort of decomposing it. Consider the second sublanguage (from the left). It is easy to see that $\overline{x} \xrightarrow{r_2(r_1r_3r_5)} \overline{x}' \;\Rightarrow\; \overline{x} \xrightarrow{(r_1r_3r_5)r_2} \overline{x}'$, so by stratification $(r_1r_3r_5+r_2)^*$ is subsumed by $(r_1r_3r_5)^*r_2^*$. Furthermore, $r_2^*(r_1+r_2+r_4r_5+r_4r_6)^* = (r_1+r_2+r_4r_5+r_4r_6)^*$, that is r_2^* is "swallowed up" by the next sub language. Finally, we have already seen that $r_1r_3r_5$ does not yield anything new, so the fixpoint is expressed by: $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2+r_4r_5+r_4r_6)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$. Choose $R = \{r_4r_5, r_4r_6\}$. Since $\xi_1(\overline{x}) \;\Rightarrow\; x_4 = 0$, we get(after deleting the zero clause $r_2r_4r_6$):

$$\begin{array}{rrrrrrrll}
0 & -1 & 1 & 0 & 0 & 0 & -1 & x_2 > 0, x_7 > 0, x_1 > 0 & : r_1 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & x_2 > 0, x_6 >, x_4 > -1, x_5 > -1 & : r_2r_4r_6 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

and $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2r_4r_6)^*(r_1+r_2)^*(r_1+r_2+r_3+r_4)^*} \overline{x}'$ describes the least fixpoint, where we note that $(r_1+r_2)^*(r_1+r_2+r_3+r_4)^* = (r_1+r_2+r_3+r_4)^*$. We have: $\overline{x} \xrightarrow{r_1(r_2r_4r_6)} \overline{x}' \;\Rightarrow\; \overline{x} \xrightarrow{(r_2r_4r_6)r_1} \overline{x}'$, so stratification applies. Since, $r_1^*(r_1+r_2+r_3+r_4)^* = (r_1+r_2+r_3+r_4)^*$, and, as above, $r_2r_4r_6$ does not yield anything to the fixpoint, we get the expression: $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4)^*} \overline{x}'$. Choose $R = \{r_3\}$. Since $\xi_1(\overline{x}) \;\Rightarrow\; x_3 = 0$, we get:

$$\begin{array}{rrrrrrrll}
0 & 0 & 0 & 0 & 1 & 0 & -1 & x_2 > 0, x_7 > 0, x_1 > 0, x_3 > -1 & : r_1r_3 \\
-1 & 0 & 0 & 1 & 0 & -1 & 0 & x_2 > 0, x_6 > 0 & : r_2 \\
1 & 0 & 0 & -1 & 1 & 0 & 0 & x_4 > 0 & : r_4 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

and $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1r_3+r_2+r_4)^*(r_1+r_2+r_4)^*} \overline{x}'$ describes the least fixpoint. Choose $R = \{r_4\}$. Since $\xi_1(\overline{x}) \;\Rightarrow\; x_4 = 0$, we get:

$$\begin{array}{rrrrrrrll}
0 & 0 & 0 & 0 & 1 & 0 & -1 & x_2 > 0, x_7 > 0, x_1 > 0, x_3 > -1 & : r_1r_3 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & x_2 > 0, x_6 > 0, x_4 > -1 & : r_2r_4 \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & &
\end{array}$$

with $\exists \overline{x} : \; \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1r_3+r_2r_4)^*(r_1r_3+r_2)^*(r_1+r_2+r_4)^*} \overline{x}'$. Both $\vartheta_{r_1r_3}(\overline{x}) \;\wedge\; B'(\overline{x}) \;\Rightarrow\; B'(\overline{x}r_1r_3)$ and $\vartheta_{r_2r_4}(\overline{x}) \;\wedge\; B'(\overline{x}) \;\Rightarrow\; B'(\overline{x}r_2r_4)$ hold, so the leading language may be dropped. Consider $(r_1 r_3 + r_2)^*$. By stratification, this is subsumed by $(r_1 r_3)^* r_2^*$. As already seen, $r_1 r_3$ does not yield anything new, and $r_2^*(r_1 + r_2 + r_4)^* = (r_1 + r_2 + r_4)^*$, so the reachability set is described by: $\exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2+r_4)^*} \overline{x}'$. Choose $R = \{r_4\}$. Since $\xi_1(\overline{x}) \Rightarrow x_4 = 0$, we get:

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

and $\exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2 r_4)^*(r_1+r_2)^*} \overline{x}'$. By stratification, $(r_1 + r_2 r_4)^*$ is subsumed by $(r_2 r_4)^* r_1^*$, and $r_1^*(r_1 + r_2)^* = (r_1 + r_2)^*$, and $r_2 r_4$ does not yield anything new. We get: $\exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{(r_1+r_2)^*} \overline{x}'$. Here, for the first time, we must use the decomposition monotonic guard (decreasing). Let us, arbitrarily, choose $x_1 > 0$ as guard and $R = \{r_1\}$. By proposition 7 we have: $\overline{x} \xrightarrow{(r_1+r_2)^*} \overline{x}' \Leftrightarrow (\overline{x} \xrightarrow{r_2^*} \overline{x}' \vee \exists \overline{x}'' : \overline{x} \xrightarrow{(r_1+r_2)^*} \overline{x}'' \xrightarrow{(r_1+r_2) r_2^*} \overline{x}' \wedge x_1'' > 0)$, where $x_1'' > 0$ is "backwards invariant" for $(r_1 + r_2)^*$. Under the assumption that $x_1 > 0$, stratification applies (that is $x_1 > 0 \wedge \overline{x} \xrightarrow{r_1 r_2} \overline{x}' \Rightarrow \overline{x} \xrightarrow{r_2 r_1} \overline{x}'$, and since $x_1' > 0$ is "backwards invariant" we get $\overline{x} \xrightarrow{(r_1+r_2)^*} \overline{x}' \wedge x_1' > 0 \Rightarrow \overline{x} \xrightarrow{r_2^* r_1^*} \overline{x}'$). We finally get: $\exists \overline{x} : \xi_1(\overline{x}) \wedge \overline{x} \xrightarrow{r_2^* + r_2^* r_1^+ r_2^*} \overline{x}'$, which consists of a simple language, so by proposition 1 it can directly be compiled into an arithmetic formula:

$$\begin{array}{l}
x_1 = 1 - x_4 \wedge \\
((x_2 = 1 \ \wedge \ x_3 = 0 \ \wedge \ x_4 \geq 0) \ \vee \\
\ (x_2 = 0 \ \wedge \ x_3 = 1 \ \wedge \ x_4 = 0)) \ \wedge \\
x_5 \geq 0 \ \wedge \ x_6 \geq 0 \ \wedge \ x_7 \geq 0 \wedge \\
x_3 + x_4 + x_5 + x_6 + x_7 = q
\end{array}$$

APPENDIX D

Our method does not only apply to Petri nets, but may also be useful for proving correctness of algorithms. Let us illustrate this by proving the correctness of the Boyer-More Majority Vote Algorithm (see [23]). This algorithm determines the element v of a list L that occurs more than half times the size of the list, if such an element exists. Let y denote the size of the list L and x the number of occurrences of the possible majority element v. Put into the form of a logic program, this algorithm leads to:

$$\begin{array}{lcll}
p_1([], v, x, y) & \leftarrow & x = 0, y = 0. & \\
p_1([u|L], v, x+1, y+1) & \leftarrow & u = v, & p_1(L, v, x, y). \\
p_1([u|L], v, x, y+1) & \leftarrow & u \neq v, 2x > y, & p_1(L, v, x, y). \\
p_1([u|L], u, x+1, y+1) & \leftarrow & u \neq v, 2x = y, & p_1(L, v, x, y).
\end{array}$$

The correctness of the program above can be stated as:

$$p_1(L, v, x, y) \wedge p_2(L, w, z) \;\Rightarrow\; (v \neq w \;\Rightarrow\; z \leq y \; div \; 2)$$

where $p_2(L, w, z)$ means that the number of occurrences of an element w in the list L is z. The predicate $p_2(L, w, z)$ is defined by:

$$\begin{array}{lcll}
p_2([], w, z) & \leftarrow & z = 0. & \\
p_2([u|L], w, z+1) & \leftarrow & u = w, & p_2(L, w, z). \\
p_2([u|L], w, z) & \leftarrow & u \neq w, & p_2(L, w, z).
\end{array}$$

By unfold/fold transformations, one can replace the conjunction $p_1(L, v, x, y) \wedge p_2(L, w, z)$ by an equivalent atom $p_3(L, v, w, x, y, z)$ defined by:

$$\begin{array}{lcll}
p_3([], v, w, x, y, z) & \leftarrow & x = 0, y = 0, z = 0. & \\
p_3([u|L], v, w, x+1, y+1, z+1) & \leftarrow & u = v, u = w, & p_3(L, v, w, x, y, z). \\
p_3([u|L], v, w, x, y+1, z+1) & \leftarrow & u \neq v, 2x > y, u = w, & p_3(L, v, w, x, y, z). \\
p_3([u|L], u, w, x+1, y+1, z+1) & \leftarrow & u \neq v, 2x = y, u = w, & p_3(L, v, w, x, y, z). \\
p_3([u|L], v, w, x+1, y+1, z) & \leftarrow & u = v, u \neq w, & p_3(L, v, w, x, y, z). \\
p_3([u|L], v, w, x, y+1, z) & \leftarrow & u \neq v, 2x > y, u \neq w, & p_3(L, v, w, x, y, z). \\
p_3([u|L], u, w, x+1, y+1, z) & \leftarrow & u \neq v, 2x = y, u \neq w, & p_3(L, v, w, x, y, z).
\end{array}$$

So the correctness of the program is now expressed as:

$$p_3(L, v, w, x, y, z) \;\Rightarrow\; (v \neq w \;\Rightarrow\; z \leq y \; div \; 2)$$

By dropping the list argument and simplifying the arithmetic constraints, we get:

$$\begin{array}{lcll}
p_4(v, w, x, y, z) & \leftarrow & x = 0, y = 0, z = 0. & \\
p_4(v, w, x+1, y+1, z+1) & \leftarrow & v = w, & p_4(v, w, x, y, z). \\
p_4(v, w, x, y+1, z+1) & \leftarrow & v \neq w, 2x > y, & p_4(v, w, x, y, z). \\
p_4(w, w, x+1, y+1, z+1) & \leftarrow & v \neq w, 2x = y, & p_4(v, w, x, y, z). \\
p_4(v, w, x+1, y+1, z) & \leftarrow & v \neq w, & p_4(v, w, x, y, z). \\
p_4(v, w, x, y+1, z) & \leftarrow & 2x > y & p_4(v, w, x, y, z). \\
p_4(u, w, x+1, y+1, z) & \leftarrow & u \neq v, 2x = y, u \neq w, & p_4(v, w, x, y, z).
\end{array}$$

It is easy to realize that the equality:

$$p_4(v,w,x,y,z) \ \Leftrightarrow \ \exists L: \ p_3(L,v,w,x,y,z)$$

holds. Therefore the proof of correctness is reduced to proving the implication:

$$p_4(v,w,x,y,z) \wedge v \neq w \ \Rightarrow \ z \leq y \ div \ 2$$

Our goal is to construct a formula $\xi(x,y,z)$ of Presburger arithmetic such that

$$\xi(x,y,z) \ \Leftrightarrow \ \exists v,w: \ p_4(v,w,x,y,z) \wedge v \neq w$$

The proof of correctness then is a matter of routine integer constraint solving. The constraints of the program defining the predicate p_4 is not of the form required for our decompositional method to be applied. Furthermore, the heads of the clauses do not only involve incrementation by constants. In order to apply our method, we must model the behaviour of the program P_4 with a program P_5 that falls within the class we treat. For $2x > y$ and $2x = y$, we introduce the variable x_A defined in such a way that $x_A = 2x - y$ holds in the least fixed point of $P_4 \cup P_5$. Then $x_A > 0$ (resp. $x_A = 0$) iff $2x > y$ (resp. $2x = y$). This yields:

$$\begin{array}{lll}
p'_4(v,w,x,y,z,,x_A) & \leftarrow & x=0, y=0, z=0, x_A = 2x-y. \\
p'_4(v,w,x+1,y+1,z+1,x_A+(2\cdot 1-1)) & \leftarrow & v=w, \\
 & & p'_4(v,w,x,y,z,x_A). \\
p'_4(v,w,x,y+1,z+1,x_A+(2\cdot 0-1)) & \leftarrow & v\neq w, x_A>0, \\
 & & p'_4(v,w,x,y,z,x_A). \\
p'_4(w,w,x+1,y+1,z+1,x_A+(2\cdot 1-1)) & \leftarrow & v\neq w, x_A=0, \\
 & & p'_4(v,w,x,y,z,x_A). \\
p'_4(v,w,x+1,y+1,z,x_A+(2\cdot 1-1)) & \leftarrow & v\neq w, \\
 & & p'_4(v,w,x,y,z,x_A). \\
p'_4(v,w,x,y+1,z,x_A+(2\cdot 0-1)) & \leftarrow & x_A>0, \\
 & & p'_4(v,w,x,y,z,x_A). \\
p'_4(u,w,x+1,y+1,z,x_A+(2\cdot 1-1)) & \leftarrow & u\neq v, x_A=0, u\neq w, \\
 & & p'_4(v,w,x,y,z,x_A).
\end{array}$$

As for Petri nets with inhibitors, $x_A = 0$ is modeled by introducing a variable x_B defined so that $x_B = 1 - x_A$ holds in the least model of $P'_4 \cup P''_4$. The construction is justified by: $x_A = 0 \ \Leftrightarrow \ x_A \geq 0 \ \wedge \ x_A \leq 0 \ \Leftrightarrow \ x_A > -1 \ \wedge \ 1 - x_A > 0 \ \Leftrightarrow \ x_A > -1 \ \wedge \ x_B > 0$. For Petri nets, the constraint $x_A = 0$, which models an inhibitor arc, occurs only in places where $x_A > -1$ is an invariant of the program. Here, x_A may take negative values, why the constraint $x_A = 0$ must be replaced by $x_A > -1 \ \wedge \ x_B > 0$. We get the program:

$$\begin{array}{lll}
p''_4(v, w, x, y, z, x_A) & \leftarrow & x = 0, y = 0, z = 0, \\
 & & x_A = 2x - y, x_B = 1 - x_A. \\
p''_4(v, w, x+1, y+1, z+1, x_A+1, x_B-1) & \leftarrow & v = w, \\
 & & p''_4(v, w, x, y, z, x_A, x_B). \\
p''_4(v, w, x, y+1, z+1, x_A-1, x_B+1) & \leftarrow & v \neq w, x_A > 0, \\
 & & p''_4(v, w, x, y, z, x_A, x_B). \\
p''_4(w, w, x+1, y+1, z+1, x_A+1, x_B-1) & \leftarrow & v \neq w, x_A > -1, x_B > 0, \\
 & & p''_4(v, w, x, y, z, x_A, x_B). \\
p''_4(v, w, x+1, y+1, z, x_A+1, x_B-1) & \leftarrow & v \neq w, \\
 & & p''_4(v, w, x, y, z, x_A, x_B). \\
p''_4(v, w, x, y+1, z, x_A-1, x_B+1) & \leftarrow & x_A > 0, \\
 & & p''_4(v, w, x, y, z, x_A, x_B). \\
p''_4(u, w, x+1, y+1, z, x_A+1, x_B-1) & \leftarrow & u \neq v, x_A > -1, x_B > 0, \\
 & & u \neq w, \\
 & & p''_4(v, w, x, y, z, x_A, x_B).
\end{array}$$

It should be clear that the transformation of P_4 into P''_4 is purely mechanical and does not require insight. To model the beaviour of u, v and w is actually also mechanical, but a bit more complicated. The inportant point is that u, v and w do not occur in any expression involing functions (incrementations) and there is no relations connecting x, y or z with u, v and w. We introduce the variables x_C and x_D defined so that $x_C > 0 \ \Leftrightarrow \ v \neq w$ and $x_D > 0 \ \Leftrightarrow \ v = w$ hold. Then we may drop the variables u, v and w in the heads of the clauses since the correctness statement can be expressed in terms of x_C and x_D in stead. We get the program:

$$\begin{array}{lcl}
p_5(x,y,z,x_A,x_B,x_C,x_D) & \leftarrow & x=0, y=0, z=0, \\
 & & x_A = 2x - y, \\
 & & x_B = 1 - x_A, \\
 & & ((x_C = 0, x_D = 1)\vee \\
 & & \ (x_C = 1, x_D = 0)). \\
p_5(x+1,y+1,z+1,x_A+1,x_B-1,x_C,x_D) & \leftarrow & x_D > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x,y+1,z+1,x_A-1,x_B+1,x_C,x_D) & \leftarrow & x_C > 0, x_A > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x+1,y+1,z+1,x_A+1,x_B-1,x_C-1,x_D+1) & \leftarrow & x_C > 0, x_A > -1, x_B > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x+1,y+1,z,x_A+1,x_B-1,x_C,x_D) & \leftarrow & x_C > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x,y+1,z,x_A-1,x_B+1,x_C,x_D) & \leftarrow & x_A > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x+1,y+1,z,x_A+1,x_B-1,x_C,x_D) & \leftarrow & x_C > 0, x_A > -1, x_B > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D). \\
p_5(x+1,y+1,z,x_A+1,x_B-1,x_C+1,x_D-1) & \leftarrow & x_D > 0, x_A > -1, x_B > 0, \\
 & & p_5(x,y,z,x_A,x_B,x_C,x_D).
\end{array}$$

The correctness statement is now expressed as:

$$p_5(x,y,z,x_A,x_B,x_C,x_D) \wedge x_C > 0 \ \Rightarrow \ z \leq y \ div \ 2$$

It remains to construct a a Presburger fomula $\xi(x,y,z)$ such that

$$\xi(x,y,z) \ \Leftrightarrow \ \exists x_A, x_B, x_C, x_D : \ p_5(x,y,z,x_A,x_B,x_C,x_D) \wedge x_C > 0$$

The incrementation matrix of P_5 has the following appearance:

$$\begin{array}{ccccccclr}
1 & 1 & 1 & 1 & -1 & 0 & 0 & x_D > 0 & : r_1 \\
0 & 1 & 1 & -1 & 1 & 0 & 0 & x_A > 0, x_C > 0 & : r_2 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & x_A > -1, x_B > 0, x_C > 0 & : r_3 \\
1 & 1 & 0 & 1 & -1 & 0 & 0 & x_C > 0 & : r_4 \\
0 & 1 & 0 & -1 & 1 & 0 & 0 & x_A > 0 & : r_5 \\
1 & 1 & 0 & 1 & -1 & 0 & 0 & x_A > -1, x_B > 0, x_C > 0 & : r_6 \\
1 & 1 & 0 & 1 & -1 & 1 & -1 & x_A > -1, x_B > 0, x_D > 0 & : r_7 \\
x & y & z & x_A & x_B & x_C & x_D & &
\end{array}$$

here we immediately see that r_6 is redundant since the constraint of r_6 implies the constraint of r_4, and the heads of the clauses are identical. We get:

$$\begin{array}{ccccccclr}
1 & 1 & 1 & 1 & -1 & 0 & 0 & x_D > 0 & : r_1 \\
0 & 1 & 1 & -1 & 1 & 0 & 0 & x_A > 0, x_C > 0 & : r_2 \\
1 & 1 & 1 & 1 & -1 & -1 & 1 & x_A > -1, x_B > 0, x_C > 0 & : r_3 \\
1 & 1 & 0 & 1 & -1 & 0 & 0 & x_C > 0 & : r_4 \\
0 & 1 & 0 & -1 & 1 & 0 & 0 & x_A > 0 & : r_5 \\
1 & 1 & 0 & 1 & -1 & 1 & -1 & x_A > -1, x_B > 0, x_D > 0 & : r_7 \\
x & y & z & x_A & x_B & x_C & x_D & &
\end{array}$$

Our task is now to decompose the language $(r_1 + r_2 + r_3 + r_4 + r_5 + r_7)^*$ in order to find a linear arithmetic expression for

$$\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5+r_7)^*} \overline{x}'$$

However, it turns out that no decomposition rule applies to this latter matrix. As was seen in the example of the reader writers protocol, the least fixpoint was generated with only a subset of the clauses in the program. This suggests the heuristic to simply drop some clause of the program and hope that the fixpoint is generated any way. We choose here to remove r_7. The rationale is that (from the transformation of the original program) we know that x_C and x_D models esentially a two state automaton representing $v = w$ and $v \neq w$. The clauses r_3 and r_7 goes from one to the other, and the base case contains a disjunction of both. From the way P_5 is encoded we know that $(x_C = 1 \wedge x_D = 0) \vee (x_C = 0 \wedge x_D = 1)$ is an invariant of the program. Secondly, for wariables x, y, z, x_A and x_B, the incrementation by r_4 is identical to r_7, while r_4 keeps x_C and x_D invariant. Therefore there is some chance that r_7 will turn out to be redundant. Note that, since $(r_1 + r_2 + r_3 + r_4 + r_5 + r_7)^* = (r_1 + r_2 + r_3 + r_4 + r_5)^*(r_1 + r_2 + r_3 + r_4 + r_5 + r_7)^*$, we have

$$\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5+r_7)^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5)^*(r_1+r_2+r_3+r_4+r_5+r_7)^*} \overline{x}'$$

By decomposing the leftmost language, if we are lucky, the fixpoint is generated. Otherwise one may continue by selecting some other subset of clauses. Clearly, such a process can not be guaranteed to succed. It turns out that for this program, the clauses r_1, r_2, r_3, r_4 and r_5 are sufficient for generating the fixpoint. Our system automatically decomposes the language $(r_1 + r_2 + r_3 + r_4 + r_5)^*$ into a flat language L such that $\overline{x} \xrightarrow{(r_1+r_2+r_3+r_4+r_5)^*} \overline{x}' \Leftrightarrow \overline{x} \xrightarrow{L} \overline{x}'$, defined by:

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

With the body, $B(\overline{x})$, of the base clause defined by:

$$\begin{array}{rcl}
B(\overline{x}) & \Leftrightarrow & x = 0 \wedge y = 0 \wedge z = 0 \wedge x_A = 2x - y \wedge x_B = 1 - x_A \wedge \\
 & & ((x_C = 0 \wedge x_D = 1) \vee (x_C = 1 \wedge x_D = 0))
\end{array}$$

The in the least fixpoint of P_5 we have:

$$\begin{array}{rcl} p_5(x,y,z,x_A,x_B,x_C,x_D) & \Leftrightarrow & \exists \overline{x}' : \ B(\overline{x}') \wedge \overline{x}' \xrightarrow{L} \langle x,y,z,x_A,x_B,x_C,x_D \rangle \\ & \Leftrightarrow & (2x - y = x_A = 1 - x_B \geq 0 \ \wedge \ y \geq x \geq z \geq 0 \ \wedge \\ & & \ x_C = 0 \ \wedge \ x_D = 1) \\ & & \qquad\qquad \vee \\ & & (2x - y = x_A = 1 - x_B \geq 0 \ \wedge \ y \geq z + x \ \wedge \\ & & \ y \geq 0 \ z \geq 0 \ \wedge \ x_C = 1 \ \wedge \ x_D = 0) \end{array}$$

and we get:

$$\begin{array}{rcl} \xi(x,y,z) & \Leftrightarrow & \exists x_A, x_B, x_C, x_D : \ p_5(x,y,z,x_A,x_B,x_C,x_D) \ \wedge \ x_C > 0 \\ & \Leftrightarrow & (2x \geq y \geq z + x \ \wedge \ y \geq 0 \ z \geq 0) \end{array}$$

It is easy to verify that

$$\xi(x,y,z) \ \Leftrightarrow \ z \leq y \ div 2$$

holds.

APPENDIX E: Program Code

In this appendix we give the code for the program we used for the decomposition of the readers writers protocol and the Boyer-Moore Majority Vote Algorithm.

```
%==============================================================================
%
%    Sets and lists
%

member([E|_],E) :- !.
member([_|L],E) :-
         member(L,E).

append([],L2,L2).
append([A|L1],L2,[A|L3]) :-
         append(L1,L2,L3).

conc_seq(s(L1),s(L2),s(L3)) :-
       append(L1,L2,L3).

%======================================
%  nth(N,List,Element).
%
%           Returns the N:th Element of List. The first element is number 1.

nth(1,[Element|_],Element) :- !.
nth(N,[_|List],Element) :- !,
      N > 1,
      N1 is N - 1,
      nth(N1,List,Element).

%======================================
%  delete_nth(N,List,List2).
%
%           Deletes the N:th Element of List. The first element is number 1.

delete_nth(1,[_|L],L) :- !.
delete_nth(N,[E|List],[E|List2]) :- !,
      N > 1,
      N1 is N - 1,
      delete_nth(N1,List,List2).

%======================================
%  subset(L1,L2).
```

```
%

subset([],_).
subset([E|L1],L2) :-
      member(L2,E),!,
      subset(L1,L2).

%=======================================
%  delete(List,Element,List_delete).
%
%           Removes the first occurence of Element in List
%

delete([],_,[]).
delete([Element|List],Element,List) :- !.
delete([X|List],Element,[X|List_delete]) :- !,
      delete(List,Element,List_delete).

%========================================
%  union(L1,L2,L_union)
%

union([],L2,L2).
union([E|L1],L2,L_union) :-
      member(L2,E),!,
      union(L1,L2,L_union).
union([E|L1],L2,[E|L_union]) :-
      union(L1,L2,L_union).

%=======================================
%  member_delete(List,Element,List_delete).
%
%           If Element is a member of List, then success and List_delete is
%           List with the first occurrence of Element removed.
%

member_delete([Element|List],Element,List) :- !.
member_delete([X|List],Element,[X|List_delete]) :- !,
      member_delete(List,Element,List_delete).

%=======================================
%  insert(List,Element,New_list).
%
%           Inserts Element in List if List does not allready contain Element.
%
```

```
insert([],Element,[Element]).
insert([Element|List],Element,[Element|List]) :- !.
insert([X|List],Element,[X|New_list]) :-
      insert(List,Element,New_list).

%=======================================
%  insert_in_all(List_of_lists,Element,List_of_lists_new)
%
%           List_of_lists_new is the result of inserting Element in every
%           list in List_of_lists
%

insert_in_all([],_,[]).
insert_in_all([L|L_of_ls],E,[L_new|L_of_ls_new]) :-
      insert(L,E,L_new),
      insert_in_all(L_of_ls,E,L_of_ls_new).

%=======================================
%  diff_list(L1,L2,L_diff).
%
%           L_diff is all elements in L1 not occurring in L2
%

diff_list([],_,[]).
diff_list([E|L1],L2,L_diff) :-
      member(L2,E), !,
      diff_list(L1,L2,L_diff).
diff_list([E|L1],L2,[E|L_diff]) :-
      diff_list(L1,L2,L_diff).

%=======================================
%  cons_to_all(List_of_list,Element,List_of_lists_e).
%
%           Puts Element at the beginning of every list in List_of_lists
%

cons_to_all([],_,[]).
cons_to_all([L|L_of_ls],E,[[E|L]|L_of_ls_e]) :-
      cons_to_all(L_of_ls,E,L_of_ls_e).

%=======================================
%  power_set(L,Pow)
%
%           Returns the power set of L, except the empty list.
```

```
%

power_set(L,Pow) :-
      length(L,N),
      (N =:= 0 -> Pow = [] ;
        power_set(1,N,L,Pow)).

power_set(N,N,L,[L]) :- !.
power_set(M,N,L,Pow) :-
      M1 is M + 1,
      power_set(M1,N,L,Pow1),
      all_of_lnth(M,L,Pow2),
      append(Pow2,Pow1,Pow).

%========================================
%  all_of_lnth(N,List,Sub_n).
%
%            Succeeds if N is less than or equal to the length of List.
%            Sub_n is all sublists of length N.
%

all_of_lnth(0,_,[[]]).
all_of_lnth(N,L,[L]) :-
      length(L,N), !.
all_of_lnth(N,[E|L],Sub_n) :-
      N >= 0,
      N1 is N - 1,
      all_of_lnth(N1,L,Sub_n1),
      all_of_lnth(N,L,Sub_n2),
      cons_to_all(Sub_n1,E,Sub_n1_e),
      append(Sub_n1_e,Sub_n2,Sub_n).

nth_subset(L1,0,[],L1) :- !.
nth_subset([E|L1],N,[E|L2],CL2) :-
      (N mod 2) =:= 1,!,
      N1 is N // 2,
      nth_subset(L1,N1,L2,CL2).
nth_subset([E|L1],N,L2,[E|CL2]) :-
      N1 is N // 2,
      nth_subset(L1,N1,L2,CL2).

max_no_subsets(List,N) :-
      length(List,L),
      N is exp(2,L) - 1.
```

```
max_triple(L1,L2,L3,[Max1,Max2,Max3]) :-
      max_no_subsets(L1,Max1),!,
      max_no_subsets(L2,Max2),!,
      max_no_subsets(L3,Max3),!.

triples_of_subsets(L1,L2,L3,[N1,N2,N3],Sub1,CSub1,Sub2,Sub3) :-
      nth_subset(L1,N1,Sub1,CSub1),!,
      nth_subset(L2,N2,Sub2,_),!,
      nth_subset(L3,N3,Sub3,_),!.

num_triples_enum([Max1,Max2,Max3],[N1,N2,N3],[N1_next,N2_next,N3_next]) :-
      (N3 < Max3 -> (N1 =< Max1, N2 =< Max2,
                     N1_next is N1, N2_next is N2, N3_next is N3 +1) ;
      (N2 < Max2 -> (N1 =< Max1, N3 =< Max3,
                     N1_next is N1, N2_next is N2 + 1, N3_next is 0) ;
                    (N1 < Max1, N2 =< Max2, N3 =< Max3,
                     N1_next is N1 + 1, N2_next is 0, N3_next is 0))).

next_subset_triple(L1,L2,L3,Max,Triple,Next_triple,Sub1,Csub1,Sub2,Sub3) :-
      num_triples_enum(Max,Triple,Next_triple),!,
      triples_of_subsets(L1,L2,L3,Next_triple,Sub1,Csub1,Sub2,Sub3),!.


%============================================================================
%
%     Arithmetic
%

plus(X,Y,Z) :-
      number(X),
      number(Y),
      Z is X + Y.
plus('infty',Y,'infty') :-
      Y \== '-infty'.
plus(X,'infty','infty') :-
      X \== '-infty'.
plus('-infty',Y,'-infty') :-
      Y \== 'infty'.
plus(X,'-infty','-infty') :-
      X \== 'infty'.


minus(X,Y,Z) :-
      number(X),
      number(Y),
      Z is X - Y.
minus('infty',Y,'infty') :-
      Y \== 'infty'.
minus(X,'infty','-infty') :-
      X \== 'infty'.
minus('-infty',Y,'-infty') :-
      Y \== '-infty'.
minus(X,'-infty','infty') :-
      X \== '-infty'.


geq('infty',_).
geq(_,'-infty').
geq(X,Y) :-
      number(X),
      number(Y),
      X >= Y.


maximum(X,Y,Max) :-
      (geq(X,Y) -> Max = X ; Max = Y).

minimum(X,Y,Min) :-
      (geq(X,Y) -> Min = Y ; Min = X).

vec_add([],[],[]).
vec_add([A|V1],[B|V2],[C|V3]) :-
      plus(A,B,C),
      vec_add(V1,V2,V3).

vec_subtr([],[],[]).
vec_subtr([A|V1],[B|V2],[C|V3]) :-
      minus(A,B,C),
      vec_subtr(V1,V2,V3).

vec_eq([],[]).
vec_eq([V|V1],[V|V2]) :-
      vec_eq(V1,V2).

vec_geq([],[]).
vec_geq([A|V1],[B|V2]) :-
      geq(A,B),!,
      vec_geq(V1,V2).

minimal([],_).
minimal([Vec1|Vec_set],Vec) :-
      (vec_geq(Vec,Vec1) -> vec_eq(Vec,Vec1) ; true),!,
      minimal(Vec_set,Vec).

maximal([],_).
maximal([Vec1|Vec_set],Vec) :-
      (vec_geq(Vec1,Vec) -> vec_eq(Vec,Vec1) ; true),!,
      maximal(Vec_set,Vec).

get_first_minimal([Vec],Vec) :- !.
get_first_minimal([Vec|Vecs],Vec) :-
      minimal(Vecs,Vec),!.
get_first_minimal([_|Vecs],Vec) :-
      get_first_minimal(Vecs,Vec).

get_first_maximal([Vec],Vec) :- !.
get_first_maximal([Vec|Vecs],Vec) :-
      maximal(Vecs,Vec),!.
get_first_maximal([_|Vecs],Vec) :-
      get_first_maximal(Vecs,Vec).

delete_all_greater_than([],_,[]).
delete_all_greater_than([Vec|Vecs],Vec1,Vecs2) :-
      vec_geq(Vec1,Vec),!,
      delete_all_greater_than(Vecs,Vec1,Vecs2).
delete_all_greater_than([Vec|Vecs],Vec1,[Vec|Vecs2]) :-
      delete_all_greater_than(Vecs,Vec1,Vecs2).

delete_all_less_than([],_,[]).
delete_all_less_than([Vec|Vecs],Vec1,Vecs2) :-
      vec_geq(Vec,Vec1),!,
      delete_all_less_than(Vecs,Vec1,Vecs2).
delete_all_less_than([Vec|Vecs],Vec1,[Vec|Vecs2]) :-
      delete_all_less_than(Vecs,Vec1,Vecs2).

lub([],[],[]).
lub([A|V1],[B|V2],[C|V3]) :-
      maximum(A,B,C),
      lub(V1,V2,V3).

big_lub([V],V) :- !.
big_lub([V|Vecs],Vlub) :-
      big_lub(Vecs,Vlub1),
      lub(V,Vlub1,Vlub),!.
```

```
all_the_same([],_).
all_the_same([C|Vec],C) :-
      all_the_same(Vec,C).

make_all_the_same([],_,[]).
make_all_the_same([_|Vec1],C,[C|Vec2]) :-
     make_all_the_same(Vec1,C,Vec2).

null_vec(Vec) :-
      all_the_same(Vec,0).

infty_vec(Vec) :-
      all_the_same(Vec,'-infty').

%========================================
%  all_possible_sums(N,Coeff,0,Sum_list)
%

all_possible_sums(1,C,_,[[C]]).
all_possible_sums(N,C,C_av,Sum_list) :-
      C > 0,
      N > 1,!,
      N1 is N - 1,
      all_possible_sums(N1,C_av,0,Sums1),
      cons_to_all(Sums1,C,Sums2),
      C1 is C - 1,
      C_av1 is C_av + 1,
      all_possible_sums(N,C1,C_av1,Sums3),
      append(Sums2,Sums3,Sum_list).
all_possible_sums(N,0,C_av,Sum_list) :-
      N > 1,!,
      N1 is N - 1,
      all_possible_sums(N1,C_av,0,Sums1),
      cons_to_all(Sums1,0,Sum_list).

%=================================================================

imply(A,B) :-
      vec_geq(A,B).

%=================================================================
%
%        Primitives
%
```

```
create_guard(N,Nmax,_,[]) :-
      N > Nmax, !.
create_guard(N,Nmax,Guard_vars,[0|Guard_vec]) :-
      variable_argument(G_var,N),
      member(Guard_vars,G_var),
      N1 is N + 1,
      create_guard(N1,Nmax,Guard_vars,Guard_vec).
create_guard(N,Nmax,Guard_vars,['-infty'|Guard_vec]) :-
      N1 is N + 1,
      create_guard(N1,Nmax,Guard_vars,Guard_vec).

create_rule(Number_of_arguments,Name,Coefficients,Guard_variables,Rule) :-
      create_guard(1,Number_of_arguments,Guard_variables,Guard),
      Rule = [s([Name]),Coefficients,Guard].

create_rules([],[],_).
create_rules([[Rn,Coeffs|Guards]|Rules_to_be],[Rule|Rules],N_of_args) :-
      length(Coeffs,N_of_args),
      create_rule(N_of_args,Rn,Coeffs,Guards,Rule),!,
      create_rules(Rules_to_be,Rules,N_of_args).

get_rulename([R_name|_],R_name).

get_coefficients([_,Coeffs|_],Coeffs).

get_guard([_,_,Guard],Guard).

conc_paths([S1,Coeff1,Guard1],[S2,Coeff2,Guard2],
          [S,Coeff,Guard]) :-
      conc_seq(S1,S2,S),
      vec_add(Coeff1,Coeff2,Coeff),
      vec_subtr(Guard2,Coeff1,C),
      lub(Guard1,C,Guard).

delete_guards_from_rule([Rn,Coeffs,Guard_vec],Guard_pos,[Rn,Coeffs,Guard2]) :-
      mask_guard(Guard_vec,Guard_pos,Guard2).

mask_guard(Guard,Guard_pos,Guard2):-
      mask_guard(Guard,Guard_pos,Guard2,1).

mask_guard([],_,[],_).
mask_guard([_|Guard],Guard_pos,['-infty'|Guard2],N) :-
      member(Guard_pos,N),!,
      N1 is N + 1,
      mask_guard(Guard,Guard_pos,Guard2,N1).
mask_guard([G|Guard],Guard_pos,[G|Guard2],N) :-
      N1 is N + 1,
      mask_guard(Guard,Guard_pos,Guard2,N1).

get_guards_of_rules([],[]).
get_guards_of_rules([[_,_,Guard]|Rules],Guards) :-
      get_guards_of_rules(Rules,Guards1),
      insert(Guards1,Guard,Guards).

first_non_infty([],[],[]).
first_non_infty(['-infty'|Guard],['-infty'|Atomic_guard],['-infty'|Guard2]) :-
      !,
      first_non_infty(Guard,Atomic_guard,Guard2).
first_non_infty([G|Guard],[G|Atomic_guard],['-infty'|Guard]) :-
      make_all_the_same(Guard,'-infty',Atomic_guard).

split_into_atomics(Guard,[]) :-
      infty_vec(Guard),!.
split_into_atomics(Guard,[Atomic_guard|Atomic_guards]) :-
      first_non_infty(Guard,Atomic_guard,Guard1),
      split_into_atomics(Guard1,Atomic_guards).

split_all_into_atomics([],[]).
split_all_into_atomics([Guard|Guards],Atomic_guards) :-
      split_all_into_atomics(Guards,Atomic_guards1),
      split_into_atomics(Guard,Atomic_guards2),
      union(Atomic_guards1,Atomic_guards2,Atomic_guards).

subsumed_by([_,Coeff1,Guard1],[_,Coeff2,Guard2]) :-
      vec_eq(Coeff1,Coeff2),
      vec_geq(Guard2,Guard1).

perm_both([_,Coeff1,Guard1],[_,Coeff2,Guard2]) :-
      vec_subtr(Guard2,Coeff1,Hyp1),!,
      lub(Hyp1,Guard1,Hyp),!,
      vec_geq(Hyp,Guard2),!,
      vec_subtr(Guard1,Coeff2,Concl),!,
      vec_geq(Hyp,Concl).

i_perm_both(I,[_,Coeff1,Guard1],[_,Coeff2,Guard2]) :-
      vec_subtr(Guard2,Coeff1,Hyp1),!,
      lub(Hyp1,Guard1,Hyp2),!,
      lub(Hyp2,I,Hyp),!,
      vec_geq(Hyp,Guard2),!,
      vec_subtr(Guard1,Coeff2,Concl),!,
      vec_geq(Hyp,Concl).

i_pos_inv(I,[_,Coeff|_]) :-
      vec_subtr(I,Coeff,Concl),!,
      vec_geq(I,Concl),!.

i_neg_inv(I,[_,Coeff|_]) :-
      vec_subtr(I,Coeff,Hyp),!,
      vec_geq(Hyp,I),!.

guard_assume_i([],[],[]).
guard_assume_i([G|Guard],[C|I],[G1|Guard1]) :-
      (geq(C,G) -> G1 = '-infty' ; G1 = G),!,
      guard_assume_i(Guard,I,Guard1).

reduce_rule_by_i([Rn,Coeffs,Guard],I,[Rn,Coeffs,Guard1]) :-
      guard_assume_i(Guard,I,Guard1).

%=================================================================

delete_guard_from_rule_n(1,[Rule|Rules],G,[Rule2|Rules]) :- !,
      variable_argument(G,G_pos),
      delete_guards_from_rule(Rule,[G_pos],Rule2).
delete_guard_from_rule_n(N,[Rule|Rules],G,[Rule|Rules2]) :-
      N > 1,
      N1 is N -1,
      delete_guard_from_rule_n(N1,Rules,G,Rules2).

permboth_all([],_).
permboth_all([Rule1|Rules1],Rules2) :-
      permboth_all_1(Rules2,Rule1),!,
      permboth_all(Rules1,Rules2).

permboth_all_1([],_).
permboth_all_1([Rule2|Rules2],Rule1) :-
      perm_both(Rule1,Rule2),!,
      permboth_all_1(Rules2,Rule1).

i_permboth_all([],_,_) :- !.
i_permboth_all([Rule1|Rules1],Rules2,I) :-
      i_permboth_all_1(Rules2,Rule1,I),!,
      i_permboth_all(Rules1,Rules2,I),!.

i_permboth_all_1([],_,_) :- !.
i_permboth_all_1([Rule2|Rules2],Rule1,I) :-
```

```
      i_perm_both(I,Rule1,Rule2),!,
      i_permboth_all_1(Rules2,Rule1,I),!.

monotonic_i([Rule],I,Sign) :- !,
      ((Sign = pos, i_pos_inv(I,Rule)) ;
       (Sign = neg, i_neg_inv(I,Rule))).
monotonic_i([Rule|Rules],I,Sign) :-
      ((Sign = pos, i_pos_inv(I,Rule)) ;
       (Sign = neg, i_neg_inv(I,Rule))),
      monotonic_i(Rules,I,Sign).

rule_makes_all_i_inv(_,[],_).
rule_makes_all_i_inv(Rule,[I|Is],Sign) :-
      ((Sign = pos, i_pos_inv(I,Rule)) ;
       (Sign = neg, i_neg_inv(I,Rule))),
      rule_makes_all_i_inv(Rule,Is,Sign).

all_rules_i_pos_inv([],_).
all_rules_i_pos_inv([Rule|Rules],I) :-
      i_pos_inv(I,Rule),!,
      all_rules_i_pos_inv(Rules,I).

all_rules_i_neg_inv([],_).
all_rules_i_neg_inv([Rule|Rules],I) :-
      i_neg_inv(I,Rule),!,
      all_rules_i_neg_inv(Rules,I).

i_in_rule(I,Rule) :-
      get_guard(Rule,Guard),!,
      imply(Guard,I).

get_rules_of_i([],_,[]).
get_rules_of_i([Rule|Rules],I,[Rule|Rules1]) :-
      i_in_rule(I,Rule),!,
      get_rules_of_i(Rules,I,Rules1).
get_rules_of_i([_|Rules],I,Rules1) :-
      get_rules_of_i(Rules,I,Rules1).

remove_rules_of_i([],_,[]).
remove_rules_of_i([Rule|Rules],I,Rules1) :-
      i_in_rule(I,Rule),!,
      remove_rules_of_i(Rules,I,Rules1).
remove_rules_of_i([Rule|Rules],I,[Rule|Rules1]) :-
      remove_rules_of_i(Rules,I,Rules1).
```

```
reduce_rules_by_i([],_,[]).
reduce_rules_by_i([Rule|Rules],I,[Rule1|Rules1]) :-
      reduce_rule_by_i(Rule,I,Rule1),
      reduce_rules_by_i(Rules,I,Rules1).

realy_all_coeff_null(Rule) :-
      get_coefficients(Rule,Coeff_vec),
      null_vec(Coeff_vec).

remove_null_rules([],[],[]).
remove_null_rules([Rule|Rules],Rules_no_null,[Rule|Rules_null]) :-
      realy_all_coeff_null(Rule),!,
      remove_null_rules(Rules,Rules_no_null,Rules_null).
remove_null_rules([Rule|Rules],[Rule|Rules_no_null],Rules_null) :-
      remove_null_rules(Rules,Rules_no_null,Rules_null).

%===================================================================

%========================================

stratifyable(Rules,Rules1,Rules2) :-
      length(Rules,Nrules),!,
      Nrules >= 2,!,
      Nrules =< 15,!,
      Max is Nrules // 2,!,
      stratifyable(1,Max,Rules,Rules1,Rules2).

stratifyable(N,Max,Rules,Rules1,Rules2) :-
      N =< Max,
      all_of_lnth(N,Rules,List_of_rules),
      find_perm(Rules,List_of_rules,Rules1,Rules2),!.
stratifyable(N,Max,Rules,Rules1,Rules2) :-
      N < Max,
      N2 is N + 1,
      stratifyable(N2,Max,Rules,Rules1,Rules2).

find_perm(Rules,[Rules_test1|_],Rules1,Rules2):-
      (diff_list(Rules,Rules_test1,Rules_test2) ->
      (permboth_all(Rules_test2,Rules_test1) ->
          (Rules1 = Rules_test1,
           Rules2 = Rules_test2) ;
          (permboth_all(Rules_test1,Rules_test2),
           Rules1 = Rules_test2,
           Rules2 = Rules_test1))).
find_perm(Rules,[_|List_of_rules],Rules1,Rules2):-
```

```
      find_perm(Rules,List_of_rules,Rules1,Rules2).

%========================================

monotonic_guard(Rules,I,Rules1,Rules2,Sign) :-
      get_guards_of_rules(Rules,Guards),!,
      split_all_into_atomics(Guards,Atomic_guards),!,
      find_monotonic_guard(Rules,Atomic_guards,I,Sign),!,
      remove_rules_of_i(Rules,I,Rules31),
      reduce_rules_by_i(Rules,I,Rules32),
      (Sign = pos -> (Rules1 = Rules31, Rules2 = Rules32) ;
                     (Rules1 = Rules32, Rules2 = Rules31)).

find_monotonic_guard(Rules,Atomic_guards,I,Sign) :-
      get_first_minimal(Atomic_guards,I1),!,
      delete_all_greater_than(Atomic_guards,I1,Atomic_guards1),!,
      (monotonic_i(Rules,I1,Sign) ->
         I = I1 ;
         find_monotonic_guard(Rules,Atomic_guards1,I,Sign)).

%========================================

monotonic_rule(Rules,Rules1,Rule) :-
      get_guards_of_rules(Rules,Guards),!,
      find_monotonic_rule(Rules,Guards,Rule),!,
      delete(Rules,Rule,Rules1).

find_monotonic_rule([Rule|_],Guards,Rule) :-
      rule_makes_all_i_inv(Rule,Guards,_),!.
find_monotonic_rule([_|Rules],Guards,Rule) :-
      find_monotonic_rule(Rules,Guards,Rule).

%========================================

post_fusion_gen(Rules,Rules1,Rules2,Rules3,Rules4,I1,I2) :-
      get_guards_of_rules(Rules,Guards),!,
      split_all_into_atomics(Guards,Ato_guards),!,
      find_post_fusion_gen(Rules,Ato_guards,
                          Rules1,Rules2,Rules3,Rules4,I1,I2).

find_post_fusion_gen(Rules,[Ato_guard|Ato_guards],
                     Rules1,Rules2,Rules3,Rules4,I1,I2) :-
      get_rules_of_i(Rules,Ato_guard,Rules_i),!,
      diff_list(Rules,Rules_i,Rules_noti),!,
     (try_postfusion(Rules_noti,Rules_i,
                     Rules1,Rules2,Rules3,Rules4,I1,I2) -> true ;
      find_post_fusion_gen(Rules,Ato_guards,
                           Rules1,Rules2,Rules3,Rules4,I1,I2)).

try_postfusion(Rules_noti,Rules_i,
                       Rules1,Rules2,Rules3,Rules4,I1,I2) :-
      get_guards_of_rules(Rules_i,Guards_i),!,
      split_all_into_atomics(Guards_i,Ato_i),!,
      max_triple(Rules_i,Ato_i,Ato_i,Max),!,
      try_all_possibilities([0,1,0],Max,Ato_i,Rules_i,Rules_noti,
                          Rules1,Rules2,Rules3,Rules4,I1,I2),!.

try_all_possibilities(Triple,Max,Ato_i,Rules_i,
                      Rules_noti,Rules1,Rules2,Rules3,Rules4,I1,I2) :-
      next_subset_triple(Rules_i,Ato_i,Ato_i,Max,Triple,Next_triple,
                         Rules33,Rules22,Ato_i1,Ato_i2),!,
     ((big_lub(Ato_i1,I11),
       big_lub(Ato_i2,I22),
       post_fusable(Rules22,Rules33,Rules_i,Rules_noti,I11,I22,
                    Rules1,Rules2,Rules3,Rules4,I1,I2)) -> true ;
        try_all_possibilities(Next_triple,Max,Ato_i,Rules_i,Rules_noti,
                              Rules1,Rules2,Rules3,Rules4,I1,I2)),!.

post_fusable(Rules22,Rules33,Rules_i,Rules_noti,I11,I22,
             Rules1,Rules2,Rules3,Rules4,I1,I2) :-
      Rules22 \== [],!,
      imply(I22,I11),!,                               % condition 8
      i_permboth_all(Rules_noti,Rules22,I11),!,       % condition 1
      i_permboth_all(Rules33,Rules22,I11),!,          % condition 1
      i_permboth_all(Rules_noti,Rules33,I22),!,       % condition 5
      guards_imply_i(Rules22,I11),!,                  % condition 4
      guards_imply_i(Rules33,I22),!,                  % condition 7
      conditions_2_3(Rules_noti,Rules_i,I11,Rules_ki),!, % conditions 2 and 3
      condition_6(Rules_ki,Rules33,I22),!,            % condition 6
      I1 = I11,
      I2 = I22,
      append(Rules_noti,Rules33,Rules1),
      Rules2 = Rules22,
      Rules3 = Rules33,
      make_motifs(Rules_ki,Rules_i,I11,Rules4).

guards_imply_i([],_).
guards_imply_i([Rule|Rules],I) :-
      get_guard(Rule,Guard),!,
      imply(Guard,I),!,
      guards_imply_i(Rules,I).

condition_6([],_,_).
condition_6([[Rule_noti,Ki]|Rules_ki],Rules_i,I2) :-
      condition_6_r(Rules_i,Rule_noti,Ki,I2),!,
      condition_6(Rules_ki,Rules_i,I2).

condition_6_r([],_,_,_).
condition_6_r([Rule_i|Rules_i],Rule_noti,Ki,I2) :-
      get_coefficients(Rule_i,Coeffs_i),!,
      vec_mult(Coeffs_i,Ki,Coeffs_ki),!,
      get_coefficients(Rule_noti,Coeffs_noti),!,
      vec_add(Coeffs_noti,Coeffs_ki,Cmotif),!,
      vec_subtr(I2,Cmotif,I2c),!,
      vec_geq(I2c,I2),!,
      condition_6_r(Rules_i,Rule_noti,Ki,I2).

conditions_2_3([],_,_,[]).
conditions_2_3([Rule_noti|Rules_noti],Rules_i,I1,
                                          [[Rule_noti,Ki]|Rules_ki]) :-
      conditions_2_3_r(Rule_noti,Rules_i,I1,Ki),!,
      conditions_2_3(Rules_noti,Rules_i,I1,Rules_ki).

conditions_2_3_r(Rule_noti,Rules_i,I1,Ki) :-
      i_pos_inv(I1,Rule_noti),!,
      (i_neg_inv(I1,Rule_noti) -> Ki = 0 ;
         condition_3_r(Rules_i,Rule_noti,I1,Ki)),!.

condition_3_r([],_,_,_).
condition_3_r([Rule_i|Rules_i],Rule_noti,I1,Ki) :-
      multiple_i_equiv(Rule_i,Rule_noti,I1,Ki),!,
      condition_3_r(Rules_i,Rule_noti,I1,Ki).

multiple_i_equiv([_,Coeff_i|_],[_,Coeff_noti|_],I1,Ki) :-
      vec_subtr(I1,Coeff_noti,C_diff),!,
      i_is_multiple(C_diff,Coeff_i,Ki),!.

i_is_multiple([],[],_).
i_is_multiple(['-infty'|I],[_|Vec],M) :- !,
      i_is_multiple(I,Vec,M),!.
i_is_multiple([C|I],[V|Vec],M) :- !,
      (V =\= 0 -> M is C // V ; C =:= 0),!,
      i_is_multiple(I,Vec,M),!.

vec_mult([],_,[]).
vec_mult(['-infty'|Vec1],C,['-infty'|Vec2]) :- !,
      vec_mult(Vec1,C,Vec2),!.
vec_mult([V1|Vec1],C,[V2|Vec2]) :- !,
      V2 is C * V1,!,
      vec_mult(Vec1,C,Vec2),!.

%========================================

pre_fusion_simp(Rules,Rules1,Rules2,Rules3,Rules4,I) :-
      find_prefusion_simp(Rules,1,Rules1,Rules2,Rules3,Rules4,I).

find_prefusion_simp(Rules,N,Rules1,Rules2,Rules3,Rules4,I) :-
      nth_subset(Rules,N,Rules_noti,Rules_i_0),!,
      get_guards_of_rules(Rules_i_0,Guards_i),!,
      split_all_into_atomics(Guards_i,Ato_i),!,
     (try_all_possibl_prefus_simp(Ato_i,Rules_noti,Rules_i_0,1,
                            Rules1,Rules2,Rules3,Rules4,I) -> true ;
      N1 is N +1,!,
      find_prefusion_simp(Rules,N1,Rules1,Rules2,Rules3,Rules4,I)).


try_all_possibl_prefus_simp(Ato_i,Rules_noti,Rules_i_0,N,
                      Rules1,Rules2,Rules3,Rules4,I) :-
      nth_subset(Ato_i,N,Ato_i1,_),!,
      big_lub(Ato_i1,I1),!, % ------ What if Ato_i is empty ?????
      get_rules_of_i(Rules_i_0,I1,Rules_i),!,          % Condition 6
      diff_list(Rules_i_0,Rules_i,Rules_0),!,
      (conditions_prefus_simp(I1,Rules_noti,Rules_i_0,Rules_i,Rules_0,
                        Rules1,Rules2,Rules3,Rules4,I) -> true ;
       N1 is N + 1,!,
       try_all_possibl_prefus_simp(Ato_i,Rules_noti,Rules_i_0,N1,
                            Rules1,Rules2,Rules3,Rules4,I)).

conditions_prefus_simp(I1,Rules_noti,Rules_i_0,Rules_i,Rules_0,
                  Rules1,Rules2,Rules3,Rules4,I) :-
      permboth_all(Rules_noti,Rules_0),!,              % Condition 1
      all_rules_i_pos_inv(Rules_0,I1),!,               %
      all_rules_i_neg_inv(Rules_0,I1),!,               % Condition 2
      i_permboth_all(Rules_noti,Rules_i,I1),!,         % Condition 3
      all_rules_i_pos_inv(Rules_noti,I1),!,            % Condition 4
      fuse_all(Rules_i,Rules_noti,Rules_m),!,
      all_rules_i_pos_inv(Rules_m,I1),!,               %
      all_rules_i_neg_inv(Rules_m,I1),!,               % Condition 5
      reduce_rules_by_i(Rules_m,I1,Rules_mr),!,
      I = I1,
      Rules1 = Rules_0,
      Rules2 = Rules_noti,
      Rules3 = Rules_i_0,
      append(Rules_0,Rules_mr,Rules4),!.

%=======================================

new_fusion1(Rules,Rules1,Rules2,Rules3,I1,I2) :-
      get_guards_of_rules(Rules,Guards),!,
      split_all_into_atomics(Guards,Ato_guards),!,
      find_newfusion1(Rules,Ato_guards,Rules1,Rules2,Rules3,I1,I2).

find_newfusion1(Rules,[Ato_guard|Ato_guards],
                Rules1,Rules2,Rules3,I1,I2) :-
      get_rules_of_i(Rules,Ato_guard,Rules_i),!,      % condition 6
      diff_list(Rules,Rules_i,Rules_noti),!,
     ((find_nullrules_check_perm(Ato_guard,Rules_noti,Rules_i,
                              Rules0_noti,Rules_not0i,Ato_i),
       try_all_possibl_newfus1(Ato_guard,Ato_i,Rules0_noti,Rules_not0i,
                            Rules_noti,Rules_i,1,
                            Rules1,Rules2,Rules3,I1,I2)) -> true ;
      find_newfusion1(Rules,Ato_guards,Rules1,Rules2,Rules3,I1,I2)).

try_all_possibl_newfus1(I11,Ato_i,Rules0_noti,Rules_not0i,
                       Rules_noti,Rules_i,N,
                       Rules1,Rules2,Rules3,I1,I2) :-
      nth_subset(Ato_i,N,Ato_i2,_),!,
      (conditions_newfuse1(I11,Ato_i2,Rules0_noti,Rules_not0i,
                       Rules_noti,Rules_i,
                       Rules1,Rules2,Rules3,I1,I2) -> true ;
        ( N1 is N + 1,!,
          try_all_possibl_newfus1(I11,Ato_i,Rules0_noti,Rules_not0i,
                       Rules_noti,Rules_i,N1,
                       Rules1,Rules2,Rules3,I1,I2))),!.

conditions_newfuse1(I11,Ato_i2,Rules0_noti,Rules_not0i,
                       Rules_noti,Rules_i,
                       Rules1,Rules2,Rules3,I1,I2) :-
      big_lub(Ato_i2,I22),!,
      all_rules_i_pos_inv(Rules_noti,I22),!,          % condition 3
      lub(I11,I22,I12),!,
      i_permboth_all(Rules_not0i,Rules_i,I12),!,      % condition 2
      get_rules_of_i(Rules_i,I22,Rules_i2),!,         % condition 7
      diff_list(Rules_i,Rules_i2,Rules_i1),!,
      all_rules_i_pos_inv(Rules_i1,I22),!,            % condition 4
      fuse_all(Rules_i,Rules_not0i,Rules_m),!,
      all_rules_i_pos_inv(Rules_m,I22),!,            %
      all_rules_i_pos_inv(Rules_m,I11),!,            %
      all_rules_i_neg_inv(Rules_m,I11),!,            % condition 5
      I1 = I11,
      I2 = I22,
      append(Rules_noti,Rules_i1,Rules1),
      append(Rules_m,Rules0_noti,Rules22),
      append(Rules22,Rules_i,Rules2),
      Rules3 = Rules_not0i,!.

find_nullrules_check_perm(I,Rules_noti,Rules_i,Rules0_noti,Rules_not0i,
                          Ato_i) :-
      all_rules_i_pos_inv(Rules_noti,I),!,           % condition 3
      get_i_equiv_rules(Rules_noti,I,Rules0_noti, Rules_not0i),!,
      permboth_all(Rules_not0i,Rules0_noti),!,       % condition 1
      get_guards_of_rules(Rules_i,Guards_i),!,
      split_all_into_atomics(Guards_i,Ato_i),!.

get_i_equiv_rules([],_,[],[]).
get_i_equiv_rules([Rule|Rules],I,Rules0, Rules_not0) :-
      (i_pos_inv(I,Rule) ->
         (i_neg_inv(I,Rule) -> (Rules0 = [Rule|Rules0_1],
                                 Rules_not0 = Rules_not0_1) ;
                                (Rules0 = Rules0_1,
                                 Rules_not0 = [Rule|Rules_not0_1])) ;
         (Rules0 = Rules0_1,
          Rules_not0 = [Rule|Rules_not0_1])),!,
      get_i_equiv_rules(Rules,I,Rules0_1, Rules_not0_1),!.

%========================================

pre_fusion_gen(Rules,Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) :-
      get_guards_of_rules(Rules,Guards),!,
      split_all_into_atomics(Guards,Ato_guards),!,
      find_prefusiongen(Rules,Ato_guards,
                        Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2).

find_prefusiongen(Rules,[Ato_guard|Ato_guards],
                  Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) :-
      get_rules_of_i(Rules,Ato_guard,Rules_i),!,     % condition 8
      diff_list(Rules,Rules_i,Rules_noti),!,
     ((find_nullrules_check_perm(Ato_guard,Rules_noti,Rules_i,
                                 Rules0_noti,Rules_not0i,Ato_i),
       try_all_possibl_prefusegen(Ato_guard,Ato_i,Rules0_noti,Rules_not0i,
                             Rules_noti,Rules_i,1,Rules1,Rules2,Rules3,
                             Rules4,Rules5,I1,I2)) -> true ;
      find_prefusiongen(Rules,Ato_guards,
                        Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2)).

try_all_possibl_prefusegen(I11,Ato_i,Rules0_noti,Rules_not0i,
                          Rules_noti,Rules_i,N,
                          Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) :-
      nth_subset(Ato_i,N,Ato_i2,_),!,
      (conditions_prefusegen(I11,Ato_i2,Rules0_noti,Rules_not0i,
                         Rules_noti,Rules_i,
                         Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) -> true ;
        ( N1 is N + 1,!,
          try_all_possibl_prefusegen(I11,Ato_i,Rules0_noti,Rules_not0i,
                        Rules_noti,Rules_i,N1,
                        Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2))),!.

conditions_prefusegen(I11,Ato_i2,Rules0_noti,Rules_not0i,
                        Rules_noti,Rules_i,
                        Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) :-
      big_lub(Ato_i2,I22),!,
      all_rules_i_pos_inv(Rules_noti,I22),!,          % condition 11
      all_rules_i_neg_inv(Rules0_noti,I22),!,         % condition 9 (part of)
      get_rules_of_i(Rules_i,I22,Rules12_i),!,        % condition 13
      diff_list(Rules_i,Rules12_i,Rules1_i),!,
      all_rules_i_pos_inv(Rules1_i,I22),!,            %
      all_rules_i_neg_inv(Rules1_i,I22),!,            % condition 9 (part of)
      get_i_equiv_rules(Rules_not0i,I22,
                        Rules1_noti,Rules12_noti),!,  % condition 9
      permboth_all(Rules1_noti,Rules12_noti),!,       % condition 2
      i_permboth_all(Rules1_noti,Rules_i,I11),!,      % condition 4
      i_permboth_all(Rules12_noti,Rules1_i,I11),!,    % condition 5
      lub(I11,I22,I12),!,
      i_permboth_all(Rules12_noti,Rules12_i,I12),!,   % condition 10
      fuse_all(Rules_i,Rules_not0i,Rules_m),!,
      all_rules_i_pos_inv(Rules_m,I11),!,             %
      all_rules_i_neg_inv(Rules_m,I11),!,             % condition 7
      fuse_all(Rules12_i,Rules12_noti,Rules12_m),!,
      all_rules_i_pos_inv(Rules12_m,I22),!,           %
      all_rules_i_neg_inv(Rules12_m,I22),!,           % condition 12
      reduce_rules_by_i(Rules_m,I11,Rules_mr),!,
      I1 = I11,
      I2 = I22,
      append(Rules_noti,Rules1_i,Rules1),
      append(Rules0_noti,Rules_i,Rules2),
      append(Rules0_noti,Rules12_m,Rules_temp),
      append(Rules_temp,Rules1_i,Rules3),
      append(Rules0_noti,Rules_mr,Rules4),
      Rules5 = Rules_not0i,!.

%=====================================

redundant_rule(Rules,Rule_red,Rules_nored) :-
      (redundant1_1(Rules,Rule_red,Rules_nored) -> true ;
        length(Rules,N),!,
        N > 2,!,
        redundant2_1(Rules,Rules,Rule_red,Rules_nored)),!.

redundant1_1([Rule|Rules],Rule_red,Rules_nored) :-
      (redundant1_2(Rules,Rule,Rules_r,Rule_r) ->
          (Rules_nored = Rules_r,
           Rule_red = Rule_r) ;
          (Rules_nored = [Rule|Rules_nored1],!,
           redundant1_1(Rules,Rule_red,Rules_nored1))),!.

redundant1_2([Rule1|Rules],Rule,Rules_r,Rule_r) :-
      (subsumed_by(Rule1,Rule) ->
        Rules_r = [Rule1|Rules],
        Rule_r = Rule ;
        (subsumed_by(Rule,Rule1) ->
          Rules_r = [Rule|Rules],
          Rule_r = Rule1 ;
          Rules_r = [Rule1|Rules_r1],
          Rule_r = Rule_r1,!,
          redundant1_2(Rules,Rule,Rules_r1,Rule_r1))),!.

redundant2_1([Rule|Rules1],Rules,Rule_red,Rules_nored) :-
      delete(Rules,Rule,Rules1),!,
      (redundant2_2(Rules1,Rule) ->
         Rules_nored = Rules1,
         Rule_red = Rule ;
         redundant2_1(Rules1,Rules,Rule_red,Rules_nored)),!.

redundant2_2(Rules,Rule) :-
      all_of_lnth(2,Rules,Rule_pairs),!,
      redundant2_3(Rule_pairs,Rule),!.

redundant2_3([[Rule1,Rule2]|Rule_pairs],Rule) :-
      conc_paths(Rule1,Rule2,Rule_c1),!,
      (subsumed_by(Rule_c1,Rule) -> true ;
        conc_paths(Rule2,Rule1,Rule_c2),!,
         (subsumed_by(Rule_c2,Rule) -> true ;
           redundant2_3(Rule_pairs,Rule))),!.

%========================================

post_fusion_adv(Rules,Rules1,Rules2,Rules3,Rules4,
                      Rules5,Rules6,Rules7,Rules8,I) :-
      get_guards_of_rules(Rules,Guards),!,
      split_all_into_atomics(Guards,Ato_guards),!,
      find_postfusionadv(Rules,Ato_guards,
                        Rules1,Rules2,Rules3,Rules4,
                        Rules5,Rules6,Rules7,Rules8,I).

find_postfusionadv(Rules,[Ato_guard|Ato_guards],
                  Rules1,Rules2,Rules3,Rules4,
                  Rules5,Rules6,Rules7,Rules8,I) :-
      get_rules_of_i(Rules,Ato_guard,Rules_i),!,      % condition 5
      diff_list(Rules,Rules_i,Rules_noti),!,
     ((try_all_possib_postfuseadv(Ato_guard,Rules_i,Rules_noti,
                                  Rules1,Rules2,Rules3,Rules4,
                                  Rules5,Rules6,Rules7,Rules8,I)) -> true ;
      find_postfusionadv(Rules,Ato_guards,
                         Rules1,Rules2,Rules3,Rules4,
                         Rules5,Rules6,Rules7,Rules8,I)).

try_all_possib_postfuseadv(I1,Rules_i,Rules_noti,
                           Rules1,Rules2,Rules3,Rules4,
                           Rules5,Rules6,Rules7,Rules8,I) :-
     i_permboth_all(Rules_noti,Rules_i,I1),!,         % condition 1
     find_i_inv_rules(Rules_i,I1,Rules_i0,Rules_inot0),!,
                                                      % conditions 4,2 ...
     permpos_all(Rules_noti,Rules_i0),!,              %
     permpos_all(Rules_inot0,Rules_i0),!,             % condition 7
     all_rules_i_pos_inv(Rules_noti,I1),!,           % condition 4
     find_i_inv_rules(Rules_noti,I1,Rules_noti0,Rules_notinot0),!,
                                                      % condition 3 and 2...
     all_rules_all_guards_neg_inv(Rules_i0,Rules_noti0),!,
                                                      % condition 8
     try_all_possib_postfuseadv1(I1,Rules_i,Rules_noti,Rules_i0,Rules_inot0,
                                 Rules_noti0,Rules_notinot0,1,
                                 Rules1,Rules2,Rules3,Rules4,
                                 Rules5,Rules6,Rules7,Rules8,I),!.

try_all_possib_postfuseadv1(I1,Rules_i,Rules_noti,Rules_i0,Rules_inot0,
                             Rules_noti0,Rules_notinot0,N,
                             Rules1,Rules2,Rules3,Rules4,
                             Rules5,Rules6,Rules7,Rules8,I) :-
        nth_subset(Rules_notinot0,N,Rules_s4,Rules_s5),!,
        (conditions_postfuseadv(I1,Rules_i0,Rules_inot0,Rules_noti0,
                                Rules_i,Rules_noti,Rules_notinot0,
                                Rules_s4,Rules_s5,
                                Rules1,Rules2,Rules3,Rules4,Rules5,
                                Rules6,Rules7,Rules8,I) -> true ;
         N1 is N + 1,!,
         try_all_possib_postfuseadv1(I1,Rules_i,Rules_noti,
                                     Rules_i0,Rules_inot0,
                                     Rules_noti0,Rules_notinot0,N1,
                                     Rules1,Rules2,Rules3,Rules4,
                                     Rules5,Rules6,Rules7,Rules8,I)),!.

  conditions_postfuseadv(I1,Rules_i0,Rules_inot0,Rules_noti0,
                         Rules_i,Rules_noti,Rules_notinot0,Rules_s4,Rules_s5,
                         Rules1,Rules2,Rules3,Rules4,Rules5,
                         Rules6,Rules7,Rules8,I) :-
        fuse_all(Rules_s4,Rules_inot0,Rules_s2s4),!,
        permboth_all(Rules_s2s4,Rules_i0),!,            % condition 6...
        all_rules_i_pos_inv(Rules_s2s4,I1),!,           %
        all_rules_i_neg_inv(Rules_s2s4,I1),!,           % condition 2...
        fuse_all(Rules_s5,Rules_inot0,Rules_s2s5),!,
        fuse_all(Rules_i0,Rules_s2s5,Rules_s2s5s1),!,
        permboth_all(Rules_s2s5s1,Rules_i0),!,          % condition 6...
        all_rules_i_pos_inv(Rules_s2s5s1,I1),!,         %
        all_rules_i_neg_inv(Rules_s2s5s1,I1),!,         % condition 2...
        fuse_all(Rules_noti0,Rules_inot0,Rules_s2s3),!,
        fuse_all(Rules_s5,Rules_s2s3,Rules_s2s3s5),!,
        permboth_all(Rules_s2s3s5,Rules_i0),!,          % condition 6...
        all_rules_i_pos_inv(Rules_s2s3s5,I1),!,         %
        all_rules_i_neg_inv(Rules_s2s3s5,I1),!,         % condition 2...
        I = I1,
        Rules1 = Rules_i,
        Rules2 = Rules_noti,
        Rules3 = Rules_noti0,
        Rules4 = Rules_notinot0,
        Rules5 = Rules_i0,
        Rules6 = Rules_inot0,
        fuse_all(Rules_inot0,Rules_s4,Rules_s4s2),
        fuse_all(Rules_i0,Rules_s5,Rules_s5s1),
        fuse_all(Rules_inot0,Rules_s5s1,Rules_s5s1s2),
        fuse_all(Rules_inot0,Rules_s5,Rules_s5s2),
        append(Rules_s4s2,Rules_s5s1s2,Rules7_m1),
        append(Rules_s5s2,Rules7_m1,Rules7_m2),
        append(Rules_noti0,Rules7_m2,Rules7),
        append(Rules_i0,Rules_noti,Rules8),!.

  find_i_inv_rules([],_,[],[]) :- !.
  find_i_inv_rules([Rule|Rules],I,Rules_inv,Rules_notinv) :-
        (i_pos_inv(I,Rule) ->
             (i_neg_inv(I,Rule) ->
                   (Rules_inv = [Rule|Rules_inv1],
                    Rules_notinv = Rules_notinv1) ;
                   (Rules_inv = Rules_inv1,
                    Rules_notinv = [Rule|Rules_notinv1])) ;
             (Rules_inv = Rules_inv1,
              Rules_notinv = [Rule|Rules_notinv1])),!,
        find_i_inv_rules(Rules,I,Rules_inv1,Rules_notinv1),!.

  all_rules_all_guards_neg_inv([],_) :- !.
  all_rules_all_guards_neg_inv([Rule|Rules],Rules_neginv) :-
        get_guard(Rule,Guard),!,
        all_rules_i_neg_inv(Rules_neginv,Guard),!,
        all_rules_all_guards_neg_inv(Rules,Rules_neginv),!.

  permpos_all([],_) :- !.
  permpos_all([Rule|Rules],Rules1) :-
        permpos_all1(Rules1,Rule),!,
        permpos_all(Rules,Rules1),!.

  permpos_all1([],_) :- !.
  permpos_all1([Rule1|Rules1],Rule) :-
        permpos(Rule,Rule1),!,
        permpos_all1(Rules1,Rule),!.

  permpos([_,Coeff1,Guard1],[_,Coeff2,Guard2]) :-
        vec_subtr(Guard2,Coeff1,C1),!,
        lub(C1,Guard1,C2),!,
        vec_subtr(Guard1,Coeff2,C3),!,
        vec_geq(C2,C3),!.

  %====================================

  post_fusion_4(Rules,Rules1,Rules2,Rules3,Rules4,I) :-
        get_guards_of_rules(Rules,Guards),!,
        split_all_into_atomics(Guards,Ato_guards),!,
        find_postfusion_4(Rules,Ato_guards,
                          Rules1,Rules2,Rules3,Rules4,I).

  find_postfusion_4(Rules,[Ato_guard|Ato_guards],
                    Rules1,Rules2,Rules3,Rules4,I) :-
        get_rules_of_i(Rules,Ato_guard,R34),!,      % condition 7
        diff_list(Rules,R34,R12),!,
       ((try_all_possib_postfuse_4(Ato_guard,1,R34,R12,
                              Rules1,Rules2,Rules3,Rules4,I)) -> true ;
        find_postfusion_4(Rules,Ato_guards,
                          Rules1,Rules2,Rules3,Rules4,I)).

  try_all_possib_postfuse_4(I1,N,R34,R12,
                             Rules1,Rules2,Rules3,Rules4,I) :-
       nth_subset(R34,N,R4,R3),!,
       (try_all_possib_postfuse_4_1(I1,1,R4,R3,R12,
                             Rules1,Rules2,Rules3,Rules4,I) -> true;
        N1 is N+1,!,
        try_all_possib_postfuse_4(I1,N1,R34,R12,
                                  Rules1,Rules2,Rules3,Rules4,I)),!.

  try_all_possib_postfuse_4_1(I1,N,R4,R3,R12,
                              Rules1,Rules2,Rules3,Rules4,I) :-
        nth_subset(R12,N,R1,R2),!,
       (conditions_postfuse_4(I1,1,R4,R3,R1,R2,
                             Rules1,Rules2,Rules3,Rules4,I) -> true;
        N1 is N+1,!,
        try_all_possib_postfuse_4_1(I1,N1,R4,R3,R12,
                              Rules1,Rules2,Rules3,Rules4,I)),!.

  conditions_postfuse_4(I1,1,R4,R3,R1,R2,
                        Rules1,Rules2,Rules3,Rules4,I) :-
        subsumed_regexpr(seq(R1,seq(R2,R2)),seq(R2,seq(R1,R2))),!,   % cond 1
        subsumed_regexpr(seq(R1,seq(sum(R2,epsilon),seq(R3,R3))),
                   seq(R3,seq(R1,seq(sum(R2,epsilon),R3)))),!, % cond 2
        subsumed_regexpr(seq(R3,R2),seq(R2,R3)),!,                 % cond 3
        i_subsumed_regexpr(
                   I1,
                   seq(R1,seq(sum(R2,epsilon),seq(sum(R3,epsilon),R4))),
                   seq(R4,seq(R1,seq(sum(R2,epsilon),sum(R3,epsilon))))),!,
                                                                   % cond 4
        i_invariant_regexpr(
                   I1,
                   seq(R1,seq(sum(R2,epsilon),sum(R3,epsilon)))),!,  % cond 5
        i_equiv_regexpr(
                   I1,
                   sum(R2,seq(R1,sum(R3,R4)))),!,                   % cond 6
        construct_language(sum(R1,R2),Rules1),!,
        construct_language(sum(sum(R2,R3),R4),Rules2),!,
        construct_language(sum(R2,seq(R1,seq(sum(R2,epsilon),sum(R3,R4)))),
                           Rules_temp),!,
        reduce_rules_by_i(Rules_temp,I1,Rules3),!,
        construct_language(sum(R1,sum(R2,R3)),Rules4),!,
        I = I1,!.

  subsumed_regexpr(Regexpr1,Regexpr2) :-
        construct_language(Regexpr1,Lang1),!,
        construct_language(Regexpr2,Lang2),!,
        subsumed_lang(Lang1,Lang2),!.

  i_subsumed_regexpr(I,Regexpr1,Regexpr2) :-
        construct_language(Regexpr1,Lang1),!,
        construct_language(Regexpr2,Lang2),!,
        i_subsumed_lang(Lang1,Lang2,I),!.

  i_invariant_regexpr(I,Regexpr) :-
        construct_language(Regexpr,Lang),!,
        i_invariant_lang(Lang,I),!.

  i_equiv_regexpr(I,Regexpr) :-
        construct_language(Regexpr,Lang),!,
        i_equiv_lang(Lang,I),!.

  %==================================================================
  reachability(Rules,Table,Table,_) :-
        length(Rules,N),
        N =< 1,!.
  reachability(Rules,Table,Table,Node) :-
        length(Rules,Lnth),
        already_treated(Rules,Lnth,Table,Node1),!,
        explain_alreadytreated(Node,Node1).
                %
                %
                % Redundant rule
                %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table1],Node) :-
        print_header(Rules,Node),
       %      write('Trying: redundant rule : '),
        redundant_rule(Rules,Rule_red,Rules_nored),!,
        length(Rules,Lnth),
        explain_redundantrule(Rules,Node,Rule_red,Rules_nored),
        tree_address(Node,red,Node1),
        remove_null_rules(Rules_nored,Rules_nored1,_),
        reachability(Rules_nored1,Table,Table1,Node1).
                 %
                 %
                   % Stratify
                   %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table2],Node) :-
         %      write('stratify : '),
        stratifyable(Rules,Rules1,Rules2), !,
        length(Rules,Lnth),
        explain_stratify(Rules,Node,Rules1,Rules2),
        tree_address(Node,str1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,str2,Node2),
        reachability(Rules2,Table1,Table2,Node2).
                   %
                   %
                   % Post fusion generalized
                   %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table4],Node) :-
         %      write('post-fusion : '),
        length(Rules,Lnth), Lnth=< 20,
        post_fusion_gen(Rules,Rules1,Rules2,Rules3,Rules4,I1,I2),Rules1\==[],!,
         %      length(Rules,Lnth),
        explain_postfusiongen(Rules,Node,Rules1,Rules2,Rules3,Rules4,I1,I2),
        tree_address(Node,psf1,Node1),
        remove_null_rules(Rules1,Rules1nn,_),
        reachability(Rules1nn,Table,Table1,Node1),
        tree_address(Node,psf2,Node2),
        remove_null_rules(Rules2,Rules2nn,_),
        reachability(Rules2nn,Table1,Table2,Node2),
        tree_address(Node,psf3,Node3),
        remove_null_rules(Rules3,Rules3nn,_),
        reachability(Rules3nn,Table2,Table3,Node3),
        tree_address(Node,psm,Node4),
        remove_null_rules(Rules4,Rules4nn,_),
        reachability(Rules4nn,Table3,Table4,Node4).
                   %
                   %
                   % Monotonic rule
                   %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table2],Node) :-
        monotonic_rule(Rules,Rules1,Rule),!,
        length(Rules,Lnth),
        explain_monotonicrule(Rules,Node,Rules1,Rule),
        tree_address(Node,mr,Node1),
        reachability(Rules1,Table,Table2,Node1).
                 %
                 %
                   % Monotonic guard
                   %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table2],Node) :-
        monotonic_guard(Rules,Guard,Rules1,Rules2,Sign), !,
        length(Rules,Lnth),
        explain_monotonicguard(Rules,Node,Guard,Sign,Rules1,Rules2),
        tree_address(Node,mg1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,mg2,Node2),
        reachability(Rules2,Table1,Table2,Node2).
                   %
                   %
                   % Pre fusion simple
                   %
  reachability(Rules,Table,[[Lnth,Rules,Node]|Table4],Node) :-
         %      write('pre-fusion (simple) : '),
        pre_fusion_simp(Rules,Rules1,Rules2,Rules3,Rules4,I),!,
        length(Rules,Lnth),
        explain_prefusion(Rules,Node,Rules1,Rules2,Rules3,Rules4,I),
        tree_address(Node,prf1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,prf2,Node2),
        reachability(Rules2,Table1,Table2,Node2),
        tree_address(Node,prf3,Node3),
        reachability(Rules3,Table2,Table3,Node3),
        tree_address(Node,prm,Node4),
        reachability(Rules4,Table3,Table4,Node4).
                   %
                   %
                   % Pre fusion generalized
                   %
  creachability(Rules,Table,[[Lnth,Rules,Node]|Table5],Node) :-
         %      write('pre-fusion (generalized) : '),
        pre_fusion_gen(Rules,Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2),!,
        length(Rules,Lnth),
        explain_prefusiongen(Rules,Node,
                       Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2),
        tree_address(Node,prfg1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,prfg2,Node2),
        reachability(Rules2,Table1,Table2,Node2),
        tree_address(Node,prmg1,Node3),
        reachability(Rules3,Table2,Table3,Node3),
        tree_address(Node,prmg2,Node4),
        reachability(Rules4,Table3,Table4,Node4),
        tree_address(Node,prfg3,Node5),
        reachability(Rules5,Table4,Table5,Node5).
                 %
                 %
                 % Post fusion advanced
                 %
  creachability(Rules,Table,[[Lnth,Rules,Node]|Table8],Node) :-
         %      write('post-fusion (advanced): '),
        post_fusion_adv(Rules,Rules1,Rules2,Rules3,Rules4,
                        Rules5,Rules6,Rules7,Rules8,I),!,
        length(Rules,Lnth),
        explain_postfusionadv(Rules,Node,
                        Rules1,Rules2,Rules3,Rules4,
                        Rules5,Rules6,Rules7,Rules8,I),
        tree_address(Node,psfa1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,psfa2,Node2),
        reachability(Rules2,Table1,Table2,Node2),
        tree_address(Node,psfa3,Node3),
        reachability(Rules3,Table2,Table3,Node3),
        tree_address(Node,psfa4,Node4),
        reachability(Rules4,Table3,Table4,Node4),
        tree_address(Node,psfa5,Node5),
        reachability(Rules5,Table4,Table5,Node5),
        tree_address(Node,psma,Node7),
        reachability(Rules7,Table5,Table7,Node7),
        tree_address(Node,psfa7,Node8),
        reachability(Rules8,Table7,Table8,Node8).
                 %
                 %
                 % Post fusion 4
                 %
  creachability(Rules,Table,[[Lnth,Rules,Node]|Table4],Node) :-
         %      write('post-fusion 4: '),
        post_fusion_4(Rules,Rules1,Rules2,Rules3,Rules4,I),!,
        length(Rules,Lnth),
        explain_postfusion_4(Rules,Node,
                       Rules1,Rules2,Rules3,Rules4,I),
        tree_address(Node,pf4R1R2,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,pf4R2R3R4,Node2),
        reachability(Rules2,Table1,Table2,Node2),
        tree_address(Node,pf4m,Node3),
        reachability(Rules3,Table2,Table3,Node3),
        tree_address(Node,pf4R1R2R3,Node4),
        reachability(Rules4,Table3,Table4,Node4).
                 %
                 %
                 % New fusion 1
                 %
  creachability(Rules,Table,[[Lnth,Rules,Node]|Table3],Node) :-
         %       write('new-fusion : '),
        new_fusion1(Rules,Rules1,Rules2,Rules3,I1,I2),!,
        length(Rules,Lnth),
        explain_newfusion1(Rules,Node,Rules1,Rules2,Rules3,I1,I2),
        tree_address(Node,nf1,Node1),
        reachability(Rules1,Table,Table1,Node1),
        tree_address(Node,nf2,Node2),
        reachability(Rules2,Table1,Table2,Node2),
        tree_address(Node,nf3,Node3),
        reachability(Rules3,Table2,Table3,Node3).
                 %
                   % Failed to treat the matrix
                   %
  reachability(Rules,Table,Table2,Node) :-
        explain_failure(Rules,Node,S,Rules2,only_rel_columns),
        (S == c -> Table2 = Table ;
                    (tree_address(Node,ntr,Node1),
                    reachability(Rules2,Table,Table2,Node1))).

  %==================================================================

  tree_address(Path,Node_name,Next_child) :-
        append(Path,[Node_name],Next_child).

  already_treated(Rules,Lnth,[[Lnth,Rls_tab,Node]|_],Node) :-
        same_rules(Rules,Rls_tab),!.
  already_treated(Rules,Lnth,[_|Table],Node) :-
        already_treated(Rules,Lnth,Table,Node).

  rule_member_delete(Rule,[Rl|Rls],Rls) :-
        same_rule(Rl,Rule), !.
  rule_member_delete(Rule,[Rl|Rls],[Rl|Rls_del]) :-
        rule_member_delete(Rule,Rls,Rls_del).
```

```
same_rule([_,Coeff1,Guard1],[_,Coeff2,Guard2]) :-
      vec_eq(Coeff1,Coeff2),!,
      vec_eq(Guard1,Guard2).

same_rules([],[]).
same_rules([Rule|Rules],Rls) :-
      rule_member_delete(Rule,Rls,Rls_del),
      same_rules(Rules,Rls_del).


reach :-
      program(Matrix),
      create_rules(Matrix,Rules,_),
      reachability(Rules,[],_,[top]).

reach_general :-
      program(Matrix_general),
      create_rules_general(Matrix_general,Rules,_),
      reachability(Rules,[],_,[top]).

create_guard_general(N,Nmax,_,[]) :-
      N > Nmax, !.
create_guard_general(N,Nmax,Guard_vars,[Gnum|Guard_vec]) :-
      variable_argument(G_var,N),
      get_assoc(Guard_vars,G_var,Gnum),
      N1 is N + 1,
      create_guard_general(N1,Nmax,Guard_vars,Guard_vec).
create_guard_general(N,Nmax,Guard_vars,['-infty'|Guard_vec]) :-
      N1 is N + 1,
      create_guard_general(N1,Nmax,Guard_vars,Guard_vec).

create_rule_general(Number_of_arguments,Name,
                    Coefficients,Guard_variables,Rule) :-
      create_guard_general(1,Number_of_arguments,Guard_variables,Guard),
      Rule = [s([Name]),Coefficients,Guard].

create_rules_general([],[],_).
create_rules_general([[Rn,Coeffs|Guards]|Rules_to_be],
                     [Rule|Rules],N_of_args) :-
      length(Coeffs,N_of_args),
      create_rule_general(N_of_args,Rn,Coeffs,Guards,Rule),!,
      create_rules_general(Rules_to_be,Rules,N_of_args).

get_assoc([[Key,Item]|_],Key,Item) :- !.
get_assoc([_|Alist],Key,Item) :-
```

```
      get_assoc(Alist,Key,Item),!.

%====================================================================

explain_stratify(Rules,Node,Rules1,Rules2) :-
%       print_header(Rules,Node),
     write('

'),
      write('Stratify:

'),
      print_kleene_closure(Rules),
      write(' ==> str1.str2

'),
      write('str1 = '),
      print_kleene_closure(Rules1),
      write('
str2 = '),
      print_kleene_closure(Rules2),
      write('

').

explain_monotonicguard(Rules,Node,Guard,Sign,Rules1,Rules2) :-
%       print_header(Rules,Node),
     write('

'),
      write('Monotonic guard, '),
      (Sign == pos -> write('increasing:  ') ; write('decreasing:  ')),
      print_guard(Guard),
      write('

'),
      print_kleene_closure(Rules),
      write(' ==> mg1.mg2

'),
      write('mg1 = '),
      print_kleene_closure(Rules1),
      write('
mg2 = '),
      print_kleene_closure(Rules2),
      write('

').

explain_monotonicrule(Rules,Node,Rules1,Rule) :-
%      print_header(Rules,Node),
     write('

'),
      write('Monotonic rule:  '),
      print_rule(Rule),
      write('

'),
      print_kleene_closure(Rules),
      write(' ==> mr.('),
      print_rule(Rule),
      write(')*.mr

'),
      write('mr = '),
      print_kleene_closure(Rules1),
      write('

').

explain_prefusion(Rules,Node,Rules1,Rules2,Rules3,Rules4,I) :-
%      print_header(Rules,Node),
     write('

'),
        write('Simple pre-fusion:  I = '),
        print_guard(I),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> (prf1.prf2 + prf3).prm.prf2

'),
        write('prf1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('prf2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('prf3 = '),
        print_kleene_closure(Rules3),
        write('
'),
        write('prm = '),
        print_kleene_closure(Rules4),
        write('

').

explain_redundantrule(Rules,Node,Rule_red,Rules_red) :-
%       print_header(Rules,Node),
     write('

'),
        write('Redundant rule:  '),
        print_rule(Rule_red),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> red

'),
        write('red = '),
        print_kleene_closure(Rules_red),
        write('

').

explain_prefusiongen(Rules,Node,Rules1,Rules2,Rules3,Rules4,Rules5,I1,I2) :-
%       print_header(Rules,Node),
     write('

'),
        write('Generalized pre-fusion:  I1 = '),
        print_guard(I1),
        write('  I2 = '),
        print_guard(I2),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> prfg1.prfg2.prmg1.prmg2.prfg3

'),
        write('prfg1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('prfg2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('prmg1 = '),
        print_kleene_closure(Rules3),
        write('
'),
        write('prmg2 = '),
        print_kleene_closure(Rules4),
        write('
'),
        write('prfg3 = '),
        print_kleene_closure(Rules5),
        write('

').

explain_postfusionadv(Rules,Node,Rules1,Rules2,Rules3,Rules4,
                                 Rules5,Rules6,Rules7,Rules8,I) :-
%       print_header(Rules,Node),
     write('

'),
        write('Advanced post-fusion: I = '),
        print_guard(I),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> (psfa1 + psfa2).psfa3.psfa4.psfa5.psfa6.psma.psfa7

'),
        write('psfa1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('psfa2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('psfa3 = '),
        print_kleene_closure(Rules3),
        write('
'),
        write('psfa4 = '),
        print_kleene_closure(Rules4),
        write('
'),
        write('psfa5 = '),
        print_kleene_closure(Rules5),
        write('
'),
        write('psfa6 = '),
        print_rule_sum(Rules6),
        write('
'),
        write('psma = '),
        print_kleene_closure(Rules7),
        write('
'),
        write('psfa7 = '),
        print_kleene_closure(Rules8),
        write('

').


explain_postfusiongen(Rules,Node,Rules1,Rules2,Rules3,Rules4,I1,I2) :-
%       print_header(Rules,Node),
     write('

'),
      ((Rules3 == []) -> (
        write('Simple post-fusion:  I1 = '),
        print_guard(I1),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> psf1.psf2.psm.mu''.psf1
'),
        write('psf1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('psf2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('psm = '),
        print_kleene_closure(Rules4),
        write('

')) ;  (
        write('Generalized post fusion:  I1 = '),
        print_guard(I1),
        write('   I2 = '),
        print_guard(I2),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> psf1.psf2.psf3.psm.mu''.psf1

'),
        write('psf1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('psf2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('psf3 = '),
        print_kleene_closure(Rules3),
        write('
'),
        write('psm = '),
        print_kleene_closure(Rules4),
        write('

'))).

explain_postfusion_4(Rules,Node,Rules1,Rules2,Rules3,Rules4,I) :-
%       print_header(Rules,Node),
     write('

'),
        write('Post fusion 4:  I = '),
        print_guard(I),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> pf4R1R2.pf4R2R3R4.pf4m.pf4R1R2R3''

'),
        write('pf4R1R2 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('pfR2R3R4 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('pf4m = '),
        print_kleene_closure(Rules3),
        write('
'),
        write('pf4R1R2R3 = '),
        print_kleene_closure(Rules4),
        write('

').

explain_newfusion1(Rules,Node,Rules1,Rules2,Rules3,I1,I2) :-
%       print_header(Rules,Node),
     write('

'),
        write('New-fusion:  I1 = '),
        print_guard(I1),
        write('  I2 = '),
        print_guard(I2),
        write('

'),
        print_kleene_closure(Rules),
        write(' ==> nf1.nf2.nf3
'),
        write('nf1 = '),
        print_kleene_closure(Rules1),
        write('
'),
        write('nf2 = '),
        print_kleene_closure(Rules2),
        write('
'),
        write('nf3 = '),
        print_kleene_closure(Rules3),
        write('

').

explain_alreadytreated(Node,Node1) :-
      write('----------------------------------------------------------
'),
      write('node: '),
      write(Node),
      write('

Already treated at:  '),
      write(Node1),
      write('

').

explain_failure(Rules,Node,S,Rules2,Print_mode) :-
%     print_header(Rules,Node),
     (Print_mode == whole_matrix -> print_header_whole(Rules,Node) ; true),
     write('

'),
     write('Couldn''t treat this matrix.

'),
     write('c: continue    r: remove rule    g: remove guard   '),
     write('a: show whole matrix   s: save rules   '),
     read(S1),
     (S1 == c -> (Rules2 = Rules, S = S1, write('

')) ;
         S1 == r -> (length(Rules,N),
                    write('
rule no. 1 - '),
                    write(N),
                    write(': '),
                    read(Sn),
                    delete_nth(Sn,Rules,Rules2),
                    S = S1,
                    write('

')) ;
            S1 == g ->
                   (write('guard: '),
                    read(G),
                    length(Rules,N),
                    write('
rule no. 1 - '),
                    write(N),
                    write(': '),
                    read(Sn),
                    delete_guard_from_rule_n(Sn,Rules,G,Rules2),
                    S = S1,
                    write('

')) ;
            S1 == a ->
                    explain_failure(Rules,Node,S,Rules2,whole_matrix) ;
                    write('name: '),
                    read(Name),
                    retractall(saved_rules(Name,_)),
                    assert(saved_rules(Name,Rules)),
                    explain_failure(Rules,Node,S,Rules2,not_whole_matrix)).

%==================================================================

size_as_string(Term,Size) :-
      name(Term,Charlist),
      length(Charlist,Size).

coeff_sizes_as_strings([],[]).
coeff_sizes_as_strings([Term|List],[Size|Sizes]) :-
      size_as_string(Term,Size),
      coeff_sizes_as_strings(List,Sizes).

max_coeff_sizes_as_strings([],[],[]).
max_coeff_sizes_as_strings([Term|List],[Msf|Max_so_far],[Size|Sizes]) :-
      size_as_string(Term,S),
      Size is max(Msf,S),
      max_coeff_sizes_as_strings(List,Max_so_far,Sizes).

guard_sizes_as_strings([],0,_).
guard_sizes_as_strings(['-infty'|List],Size,N) :- !,
      N1 is N + 1,
      guard_sizes_as_strings(List,Size,N1).
guard_sizes_as_strings([Term|List],Size,N) :-
      size_as_string(Term,St),
      variable_argument(Variable,N),
      size_as_string(Variable,Sv),
      N1 is N + 1,
      guard_sizes_as_strings(List,Size1,N1),
      Size is Size1 + St + Sv + 3.

var_row_cols([],[],_).
var_row_cols([_|List],[Size|Sizes],N) :-
      variable_argument(Variable,N),
      size_as_string(Variable,Size),
      N1 is N + 1,
      var_row_cols(List,Sizes,N1).

column_sizes([[_,Coeffs,Guard]],Coeff_cols,Guard_col,Guard) :-
      coeff_sizes_as_strings(Coeffs,Coeff_cols1),
      guard_sizes_as_strings(Guard,Guard_col,1),
      var_row_cols(Guard,Var_col_sizes,1),
      lub(Coeff_cols1,Var_col_sizes,Coeff_cols).
column_sizes([[_,Coeffs,Guard]|Rules],Coeff_cols,Guard_col,Relevant_cols) :-
      column_sizes(Rules,Coeff_cols1,Guard_col1,Relev_cols1),
      max_coeff_sizes_as_strings(Coeffs,Coeff_cols1,Coeff_cols),
      guard_sizes_as_strings(Guard,Guard_col2,1),
      Guard_col is max(Guard_col1,Guard_col2),
      lub(Relev_cols1,Guard,Relevant_cols).

spaces(0) :- !,
      write(' ').
spaces(N) :-
      N > 0, !,
      write(' '),
      N1 is N - 1,
      spaces(N1).

print_rule([s(Rname)|_]) :-
      print_rule_name(Rname).
print_rule_name([Rn]) :- !,
      write(Rn).
print_rule_name([Rn|Rns]) :-
      write(Rn),
      write('.'),
      print_rule_name(Rns).

print_coeffs([],[],[]).
print_coeffs([_|Coeffs],[_|Col_sizes],['-infty'|Guards]) :- !,
      print_coeffs(Coeffs,Col_sizes,Guards).
print_coeffs([C|Coeffs],[Col_size|Col_sizes],[_|Guards]) :-
      size_as_string(C,S),
      Space is Col_size - S + 1,
      spaces(Space),
      write(C),
      print_coeffs(Coeffs,Col_sizes,Guards).

print_guard(Guard) :-
      print_guard(Guard,1,no_comma).

print_guard([],_,_).
print_guard(['-infty'|Guard],N,F) :- !,
      N1 is N + 1,
      print_guard(Guard,N1,F).
print_guard([G|Guard],N,F) :-
      variable_argument(Variable,N),
      (F == comma -> write(', ') ; true),
      write(Variable),
      write('>'),
      write(G),
      N1 is N + 1,
      print_guard(Guard,N1,comma).

print_guards([],Col_size,_,_) :-
      spaces(Col_size).
print_guards(['-infty'|Guards],Col_size,N,F) :- !,
      N1 is N + 1,
      print_guards(Guards,Col_size,N1,F).
print_guards([G|Guards],Col_size,N,F) :-
      variable_argument(Variable,N),
      (F == no_comma -> write('  ') ; write(', ')),
      write(Variable),
      write('>'),
      write(G),
      N1 is N + 1,
      size_as_string(G,Sg),
      size_as_string(Variable,Sv),
      Col_size1 is Col_size - Sg - Sv - 3,
      print_guards(Guards,Col_size1,N1,comma).

print_variables([],[],_).
print_variables([_|Col_sizes],['-infty'|Guards],N) :- !,
      N1 is N + 1,
      print_variables(Col_sizes,Guards,N1).
print_variables([Col_size|Col_sizes],[_|Guards],N) :-
      variable_argument(Variable,N),
      size_as_string(Variable,S),
      Space is Col_size - S + 1,
      spaces(Space),
      write(Variable),
      N1 is N + 1,
      print_variables(Col_sizes,Guards,N1).

print_rule([s(Rn),Coeffs,Guards],Coeff_col_sizes,
                                Guard_col_size,Relevant_cols) :-
      print_coeffs(Coeffs,Coeff_col_sizes,Relevant_cols),
      write(' '),
      print_guards(Guards,Guard_col_size,1,no_comma),
      write(' : '),
      print_rule_name(Rn).

print_rules([],_,_,_).
print_rules([Rule|Rules],Coeff_col_sizes,
                         Guard_col_size,Relevant_cols) :-
      print_rule(Rule,Coeff_col_sizes,Guard_col_size,Relevant_cols),
      write('
'),
      print_rules(Rules,Coeff_col_sizes,Guard_col_size,Relevant_cols).

print_rules(Rules) :-
      column_sizes(Rules,Coeff_cols,Guard_col,Relevant_cols),
      print_rules(Rules,Coeff_cols,Guard_col,Relevant_cols),
      write('
'),
      print_variables(Coeff_cols,Relevant_cols,1),
      write('

').

print_rules_whole(Rules) :-
```

```
      column_sizes(Rules,Coeff_cols,Guard_col,Relevant_cols),
      all_cols_rel(Relevant_cols,All_cols_rel),
      print_rules(Rules,Coeff_cols,Guard_col,All_cols_rel),
      write('
'),
      print_variables(Coeff_cols,All_cols_rel,1),
      write('

').

all_cols_rel([],[]).
all_cols_rel([_|Relevant_cols],[0|All_cols_rel]) :-
      all_cols_rel(Relevant_cols,All_cols_rel).

print_header(Rules,Node) :-
      write('-----------------------------------------------------------
'),
      write('node: '),
      write(Node),
      write('

'),
      print_rules(Rules),!.

print_header_whole(Rules,Node) :-
      write('-----------------------------------------------------------
'),
      write('node: '),
      write(Node),
      write('

'),
      print_rules_whole(Rules),!.

print_kleene_closure([]) :- !,
      write('e').
print_kleene_closure(Rules) :-
      write('('),
      print_kleene_closure_1(Rules),
      write(')*').

print_kleene_closure_1([[s(Rn)|_]]) :- !,
      print_rule_name(Rn).
print_kleene_closure_1([[s(Rn)|_]|Rules]) :-
      print_rule_name(Rn),
      write(' + '),
      print_kleene_closure_1(Rules).

print_rule_sum([]) :- !,
      write('e').
print_rule_sum([[s(Rn)|_]]) :-
      print_rule_name(Rn).
print_rule_sum(Rules) :-
      write('('),
      print_kleene_closure_1(Rules),
      write(')').

%============================================================================

make_motifs([],_,_,[]).
make_motifs([[Rule,0]|Rules_ki],Rules_i,I1,Rules) :- !,
      make_motifs(Rules_ki,Rules_i,I1,Rules2),!,
      path_set_union([Rule],Rules2,Rules).
make_motifs([[Rule,Ki]|Rules_ki],Rules_i,I1,Rules) :-
      all_paths(Rules_i,Rule,Ki,Rules1),!,
      reduce_guards_ki(Rules_i,I1,Rules1,Rules11),
      make_motifs(Rules_ki,Rules_i,I1,Rules2),!,
      path_set_union(Rules11,Rules2,Rules).

all_paths(_,Rule,0,[Rule]) :- !.
all_paths(Rules_i,Rule,Ki,Rules) :-
      Ki>0,!,
      Ki1 is Ki - 1,
      all_paths(Rules_i,Rule,Ki1,Rules1),!,
      fuse_all(Rules_i,Rules1,Rules).

fuse_all([],_,[]).
fuse_all([Rule_i|Rules_i],Rules1,Rules) :-
      fuse_all(Rules_i,Rules1,Rules2),
      fuse_all_1(Rules1,Rule_i,Rules3),
      append(Rules3,Rules2,Rules).

fuse_all_1([],_,[]).
fuse_all_1([Rule1|Rules1],Rule_i,[Rule2|Rules2]) :-
      conc_paths(Rule1,Rule_i,Rule2),!,
      fuse_all_1(Rules1,Rule_i,Rules2).

path_set_union([],Paths1,Paths1).
path_set_union([Path|Paths],Paths2,Paths3) :-
      insert_path(Paths2,Path,Paths4),
      path_set_union(Paths,Paths4,Paths3).

insert_path([],Path,[Path]).
insert_path([Path1|Paths1],Path,[Path1|Paths1]) :-
      subsumed_by(Path,Path1),!.     % <----------- Fel !!!!
insert_path([Path1|Paths1],Path,Paths2) :-
      subsumed_by(Path1,Path),!,     % <----------- Fel !!!!
      insert_path(Paths1,Path,Paths2).
insert_path([Path1|Paths1],Path,[Path1|Paths2]) :-
      insert_path(Paths1,Path,Paths2).

reduce_guards_ki([Rule_i|_],I1,Rules1,Rules11) :-
      get_coefficients(Rule_i,Coeffs_i),!,
      vec_add(I1,Coeffs_i,I1c),!,
      reduce_rules_by_i(Rules1,I1c,Rules11).

%===========================================================================

perm1(Path1,Path2,P1,P2) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      construct_path(Path1,Rules,P1),!,
      construct_path(Path2,Rules,P2),!,
      perm_both(P1,P2),!.

perm2(Path1,Path2,Name,P1,P2) :-
      saved_rules(Name,Rules),!,
      construct_path(Path1,Rules,P1),!,
      construct_path(Path2,Rules,P2),!,
      perm_both(P1,P2),!.


construct_path([Rn],Rules,P) :- !,
      get_rule_with_name(Rules,Rn,P),!.
construct_path([Rn|Path],Rules,P) :- !,
      construct_path(Path,Rules,P1),!,
      get_rule_with_name(Rules,Rn,P2),!,
      conc_paths(P2,P1,P),!.

get_rule_with_name([Rule|_],Rn,Rule) :-
      get_rulename(Rule,s(Rn)),!.
get_rule_with_name([_|Rules],Rn,Rule) :-
      get_rule_with_name(Rules,Rn,Rule),!.

subsumed_path(Path1,Path2,P1,P2) :-
```

```
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      construct_path(Path1,Rules,P1),!,
      construct_path(Path2,Rules,P2),!,
      subsumed_by(P2,P1),!.

test1_reach(Paths) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      construct_paths(Paths,Rules,P_rules),!,
      reachability(P_rules,[],_,[top]).

construct_paths([],_,[]).
construct_paths([Path|Paths],Rules,[Rule|P_rules]) :-
      construct_path(Path,Rules,Rule),!,
      construct_paths(Paths,Rules,P_rules).

test2_reach(Paths,Name) :-
      saved_rules(Name,Rules),!,
      construct_paths(Paths,Rules,P_rules),!,
      reachability(P_rules,[],_,[top]).

saved_reach(Name) :-
      saved_rules(Name,Rules),!,
      reachability(Rules,[],_,[top]).


construct_language(epsilon,[epsilon]).
construct_language(sum(Reg1,Reg2),Lang) :-
      construct_language(Reg1,Lang1),
      construct_language(Reg2,Lang2),
      union(Lang1,Lang2,Lang).
construct_language(seq(Reg1,Reg2),Lang) :-
      construct_language(Reg1,Lang1),
      construct_language(Reg2,Lang2),
      concatenate_lang(Lang1,Lang2,Lang).
construct_language(Lang,Lang) :-
      lang(Lang).

lang([_|_]).

concatenate_lang([],_,[]).
concatenate_lang([Seq1|Lang1],Lang2,Lang) :-
      concatenate_lang1(Seq1,Lang2,Lang3),
      concatenate_lang(Lang1,Lang2,Lang4),
      union(Lang3,Lang4,Lang).

concatenate_lang1(_,[],[]).
concatenate_lang1(Seq1,[Seq2|Lang2],[Seq|Lang]) :-
      conc_lang_paths(Seq1,Seq2,Seq),
      concatenate_lang1(Seq1,Lang2,Lang).

conc_lang_paths(epsilon,Seq2,Seq2).
conc_lang_paths(Seq1,epsilon,Seq1).
conc_lang_paths(Seq1,Seq2,Seq) :-
      conc_paths(Seq1,Seq2,Seq).

subsumed_lang([],_) :- !.
subsumed_lang([Seq1|Lang1],Lang2) :-
      subsumed_lang1(Seq1,Lang2),!,
      subsumed_lang(Lang1,Lang2),!.

subsumed_lang1(Seq1,[Seq2|_]) :-
      subsumed_seq(Seq1,Seq2),!.
subsumed_lang1(Seq1,[_|Lang2]) :-
      subsumed_lang1(Seq1,Lang2),!.

subsumed_seq(epsilon,epsilon) :- !.
subsumed_seq([_,Coeff1,_],epsilon) :- !,
      null_veq(Coeff1),!.
subsumed_seq([_,Coeff1,Guard1],[_,Coeff2,Guard2]) :- !,
      vec_eq(Coeff1,Coeff2),!,
      vec_geq(Guard1,Guard2),!.

construct_language_names(epsilon,[epsilon]).
construct_language_names(sum(Reg1,Reg2),Lang) :-
      construct_language_names(Reg1,Lang1),
      construct_language_names(Reg2,Lang2),
      union(Lang1,Lang2,Lang).
construct_language_names(seq(Reg1,Reg2),Lang) :-
      construct_language_names(Reg1,Lang1),
      construct_language_names(Reg2,Lang2),
      concatenate_lang(Lang1,Lang2,Lang).
construct_language_names(Lang,[Rule]) :-
      name_of_rule(Lang,Rule).

name_of_rule(Name,Rule) :-
      rule_set(Rules),!,
      get_rule_with_name(Rules,Name,Rule),!.
```

```
subsumed_regexpr_names(Regexpr1,Regexpr2) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      assert(rule_set(Rules)),!,
      construct_language_names(Regexpr1,Lang1),!,
      construct_language_names(Regexpr2,Lang2),!,
      subsumed_lang(Lang1,Lang2),!,
      retractall(rule_set(_)).

i_subsumed_regexpr_names(I,Regexpr1,Regexpr2) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      assert(rule_set(Rules)),!,
      construct_language_names(Regexpr1,Lang1),!,
      construct_language_names(Regexpr2,Lang2),!,
      [[_,Coeff,_]|_] = Rules,!,
      length(Coeff,N),!,
      create_guard(1,N,I,I1),!,
      i_subsumed_lang(Lang1,Lang2,I1),!,
      retractall(rule_set(_)).

i_subsumed_lang([],_,_) :- !.
i_subsumed_lang([Seq1|Lang1],Lang2,I) :-
      i_subsumed_lang1(Seq1,Lang2,I),!,
      i_subsumed_lang(Lang1,Lang2,I),!.

i_subsumed_lang1(Seq1,[Seq2|_],I) :-
      i_subsumed_seq(Seq1,Seq2,I),!.
i_subsumed_lang1(Seq1,[_|Lang2],I) :-
      i_subsumed_lang1(Seq1,Lang2,I),!.

i_subsumed_seq(epsilon,epsilon,_) :- !.
i_subsumed_seq([_,Coeff1,_],epsilon,_) :- !,
      null_veq(Coeff1),!.
i_subsumed_seq([_,Coeff1,Guard1],[_,Coeff2,Guard2],I) :- !,
      vec_eq(Coeff1,Coeff2),!,
      lub(I,Guard1,C),!,
      vec_geq(C,Guard2),!.

i_invariant_regexpr_names(I,Regexpr) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      assert(rule_set(Rules)),!,
      construct_language_names(Regexpr,Lang),!,
      [[_,Coeff,_]|_] = Rules,!,
      length(Coeff,N),!,
      create_guard(1,N,I,I1),!,
      i_invariant_lang(Lang,I1),!,
      retractall(rule_set(_)).

i_invariant_lang([],_) :- !.
i_invariant_lang([Seq|Lang],I) :-
      i_invariant_seq(Seq,I),!,
      i_invariant_lang(Lang,I),!.

i_invariant_seq([_,Coeff,_],I) :-
      vec_subtr(I,Coeff,Ic),!,
      vec_geq(I,Ic),!.

i_equiv_regexpr_names(I,Regexpr) :-
      program(Matrix),!,
      create_rules(Matrix,Rules,_),!,
      assert(rule_set(Rules)),!,
      construct_language_names(Regexpr,Lang),!,
      [[_,Coeff,_]|_] = Rules,!,
      length(Coeff,N),!,
      create_guard(1,N,I,I1),!,
      i_equiv_lang(Lang,I1),!,
      retractall(rule_set(_)).

i_equiv_lang([],_) :- !.
i_equiv_lang([Seq|Lang],I) :-
      i_equiv_seq(Seq,I),!,
      i_equiv_lang(Lang,I),!.

i_equiv_seq([_,Coeff,_],I) :-
      vec_subtr(I,Coeff,Ic),!,
      vec_geq(I,Ic),!,
      vec_geq(Ic,I),!.
```