Deriving Filtering Algorithms from Constraint Checkers

Nicolas Beldiceanu[1], Mats Carlsson[2], and Thierry Petit[1]

[1] LINA FRE CNRS 2729, École des Mines de Nantes, FR-44307 Nantes Cedex 3, France.
{Nicolas.Beldiceanu,Thierry.Petit}@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden.
Mats.Carlsson@sics.se

April 27, 2004

SICS Technical Report T2004:08

ISRN: SICS-T–2004/08-SE

ISSN: 1100-3154

Abstract. This reportdeals with global constraints for which the set of solutions can be recognized by an extended finite automaton whose size is bounded by a polynomial in n, where n is the number of variables of the corresponding global constraint. By reformulating the automaton as a conjunction of signature and transition constraints we show how to systematically obtain a filtering algorithm. Under some restrictions on the signature and transition constraints this filtering algorithm achieves arc-consistency. An implementation based on some constraints as well as on the metaprogramming facilities of SICStus Prolog is available. For a restricted class of automata we provide a filtering algorithm for the relaxed case, where the violation cost is the minimum number of variables to unassign in order to get back to a solution. **Keywords:** Constraint Programming, Global Constraints, Filtering Algorithms, Finite Automata, Relaxation.

1 Introduction

Deriving filtering algorithms for global constraints is usually far from obvious and requires a lot of energy. As a first step toward a methodology for semi-automatic development of filtering algorithms for global constraints, Carlsson and Beldiceanu have introduced [12] an approach to design filtering algorithms by derivation from a finite automaton. As quoted in their discussion, constructing the automaton was far from obvious since it was mainly done as a rational reconstruction of an emerging understanding of the necessary case analysis related to the required pruning. However, it is commonly admitted that coming up with a checker which tests whether a ground instance is a solution or not is usually straightforward. This was for instance done for constraints defined in extension first by Vempaty [29] and later on by Amilhastre et al. [2]. This was also done for arithmetic constraints by Boigelot and Wolper [10]. Within the context of global constraints on a finite sequence of variables, the recent work of Pesant [26] uses also a finite automaton for constructing a filtering algorithm. This reportfocuses on those global constraints that can be checked by scanning once through their variables without using any extra data structure.

As a second step toward a methodology for semi-automatic development of filtering algorithms, we introduce a new approach which only requires defining a finite automaton that checks a ground instance. We extend traditional finite automata in order not to be limited only to regular expressions. Our first contribution is to show how to reformulate the automaton associated with a global constraint as a conjunction of signature and transition constraints. We characterize some restrictions on the signature and transition constraints under which the filtering algorithm induced by this reformulation achieves arc-consistency and apply this new methodology to the three following problems:

– The design of filtering algorithms for a fairly large set of global constraints.
– The design of filtering algorithms for handling the conjunction of several global constraints.
– The design of constraints for dealing with problems involving finite state systems where the transition diagram is known.

Our second contribution is to provide for a restricted class of automata a filtering algorithm for the relaxed case. This technique relies on the variable based violation cost introduced in [27, 24]. This cost was advocated as a generic way for expressing the violation of a global constraint. However, algorithms were only provided for the `soft_alldifferent` constraint [27]. We come up with an algorithm for computing a sharp bound of the minimum violation cost and with a filtering algorithm for pruning in order to avoid to exceed a given maximum violation cost.

The rest of this reportis organized as follows. Section 2 describes the kind of finite automaton used for recognizing the set of solutions associated with a global constraint. Section 3 shows how to come up with a filtering algorithm which exploits the previously introduced automaton. Section 4 describes typical applications of this technique. For a restricted class of automata, Section 5 provides a filtering algorithm for the relaxed case. Finally, the Appendix gives the different automata.

2 Description of the Automaton Used for Checking Ground Instances

We first discuss the main issues behind the task of selecting what kind of automaton to consider for expressing in a concise way the set of solutions associated with a global constraint. We consider global constraints for which any ground instance can be checked in linear time by scanning once through their variables without using any data structure. In order to concretely illustrate this point we first select a set of global constraints and write down a checker for each of them. Finally, we give for each checker a sketch of the corresponding automaton. Based on these observations, we define the type of automaton we will use.

Selecting an Appropriate Description. As we previously said, we focus on those global constraints that can be checked by scanning once through their variables. This is for instance the case of `element` [20], `minimum` [4], `pattern` [11], `global_con-`

`tiguity` [23], `lexicographic ordering` [18], `among` [6] and `inflexion` [3]. Since they illustrate key points needed for characterizing the set of solutions associated with a global constraint, our discussion will be based on the last four constraints for which we now recall the definition:

– The `global_contiguity`$(vars)$ constraint enforces for the sequence of 0-1 variables $vars$ to have at most one group of consecutive 1. For instance, the constraint `global_contiguity`$([0, 1, 1, 0])$ holds since we have only one group of consecutive 1.
– The lexicographic ordering constraint $\overrightarrow{x} \leq_{\text{lex}} \overrightarrow{y}$ over two vectors of variables $\overrightarrow{x} = \langle x_0, \ldots, x_{n-1}\rangle$ and $\overrightarrow{y} = \langle y_0, \ldots, y_{n-1}\rangle$ holds iff $n = 0$ or $x_0 < y_0$ or $x_0 = y_0$ and $\langle x_1, \ldots, x_{n-1}\rangle \leq_{\text{lex}} \langle y_1, \ldots, y_{n-1}\rangle$.
– The `among`$(nvar, vars, values)$ constraint restricts the number of variables of the sequence of variables $vars$, which take their value in a given set $values$, to be equal to the variable $nvar$. For instance, `among`$(3, [4, 5, 5, 4, 1], [1, 5, 8])$ holds since exactly 3 values of the sequence 45541 are located in $\{1, 5, 8\}$.
– The `inflexion`$(ninf, vars)$ constraint enforces the number of inflexions of the sequence of variables $vars$ to be equal to the variable $ninf$. An `inflexion` is described by one of the following patterns: a strict increase followed by a strict decrease or, conversely, a strict decrease followed by a strict increase. For instance, `inflexion`$(4, [3, 3, 1, 4, 5, 5, 6, 5, 5, 6, 3])$ holds since we can extract from the sequence 33145565563 the four subsequences 314, 565, 6556 and 563, which all follow one of these two patterns.

Parts (A1), (B1), (C1) and (D1) of Fig. 1 depict the four checkers respectively associated with `global_contiguity`, with $\leq_{\text{lex}}$, with `among` and with `inflexion`. For each checker we observe the following facts:

– Within the checker depicted by part (A1) of Fig. 1, the values of the sequence $vars[0], \ldots, vars[n-1]$ are successively compared against 0 and 1 in order to check that we have at most one group of consecutive 1. This can be translated to the automaton depicted by part (A2) of Fig. 1. The automaton takes as input the sequence $vars[0], \ldots, vars[n-1]$, and triggers successively a transition for each term of this sequence. Transitions labeled by 0, 1 and \$ are respectively associated with the conditions $vars[i] = 0$, $vars[i] = 1$ and $i = n$. Transitions leading to failure are systematically skipped. This is why no transition labeled with a 1 starts from state z.
– Within the checker given by part (B1) of Fig. 1, the components of vectors $\overrightarrow{x}$ and $\overrightarrow{y}$ are scanned in parallel. We first skip all the components that are equal and then perform a final check. This is represented by the automaton depicted by part (B2) of Fig. 1. The automaton takes as input the sequence $\langle x[0], y[0]\rangle, \ldots, \langle x[n-1], y[n-1]\rangle$ and triggers a transition for each term of this sequence. Unlike the `global_contiguity` constraint, some transitions now correspond to a condition (e.g. $x[i] = y[i]$, $x[i] < y[i]$) between two variables of the $\leq_{\text{lex}}$ constraint.
– Observe that the `among`$(nvar, vars, values)$ constraint involves a variable $nvar$ whose value is computed from a given collection of variables $vars$. The checker depicted by part (C1) of Fig. 1 counts the number of variables of $vars[0], \ldots, vars[n-$

```
global_contiguity(vars[0..n−1]):BOOLEAN
BEGIN
  i=0;
  WHILE i<n AND vars[i]=0 DO i++;
  WHILE i<n AND vars[i]=1 DO i++;
  WHILE i<n AND vars[i]=0 DO i++;
  RETURN (i=n);
END.
```

(A1)

```
≤lex(x[0..n−1],y[0..n−1]):BOOLEAN
BEGIN
  i=0;
  WHILE i<n AND x[i]=y[i] DO i++;
  RETURN (i=n OR x[i]<y[i]);
END.
```

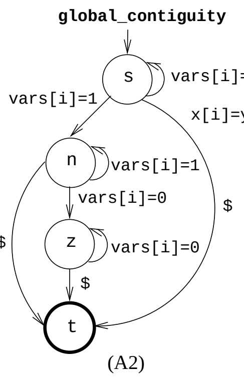
(B1)

```
among(nvar,vars[0..n−1],values):BOOLEAN
BEGIN
  i=0; c=0;
  WHILE i<n DO
    IF vars[i] in values THEN c++;
    i++;
  RETURN (nvar=c);
END.
```

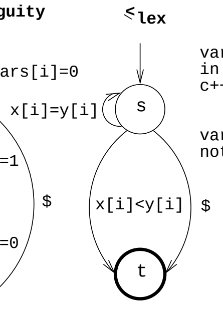
(C1)

```
inflexion(ninf,vars[0..n−1]):BOOLEAN
BEGIN
  i=0; c=0;
  WHILE i<n−1 AND vars[i]=vars[i+1] DO i++;
  IF i<n−1 THEN less=(vars[i]<vars[i+1]);
  WHILE i<n−1 DO
    IF less THEN
      IF vars[i]>vars[i+1] THEN c++; less:=FALSE;
    ELSE
      IF vars[i]<vars[i+1] THEN c++; less:=TRUE;
    i++;
  RETURN (ninf=c);
END.
```

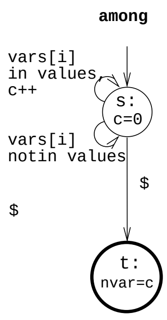
(D1)



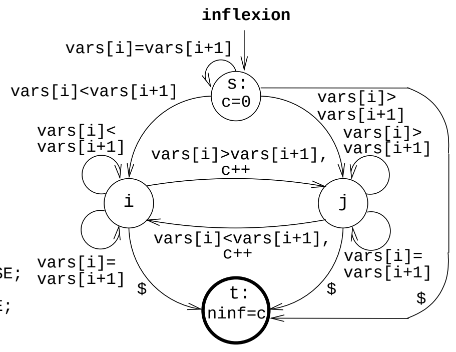
(A2)



(B2)



(C2)



(D2)

Fig. 1. Four checkers and their corresponding automata.

1] that take their value in $values$. For this purpose it uses a counter c, which is eventually tested against the value of $nvar$. This convinced us to allow the use of counters in an automaton. Each counter has an initial value which can be updated while triggering certain transitions. The final state of an automaton can enforce a variable of the constraint to be equal to a given counter. Part (C2) of Fig. 1 describes the automaton corresponding to the code given in part (C1) of the same figure. The automaton uses the counter variable c initially set to 0 and takes as input the sequence $vars[0], \dots, vars[n-1]$. It triggers a transition for each variable of this sequence and increments c when the corresponding variable takes its value in $values$. The final state returns a success when the value of c is equal to $nvar$. At this point we want to stress the following fact: It would have been possible to use an automaton which avoids the use of counters. However, this automaton would depend on the effective value of the parameter $nvar$. In addition, it would require more states than the automaton of part (C2) of Fig. 1. This is typically a problem if we want to have a fixed number of states in order to save memory as well as time.

– As the `among` constraint, the `inflexion`$(ninf, vars)$ constraint involves a variable $ninf$ whose value is computed from a given sequence of variables $vars[0], \dots, vars[n-1]$. Therefore, the checker depicted in part (D1) of Fig. 1 uses also a counter c for counting the number of inflexions, and compares its final value to the $ninf$ parameter. This program is represented by the automaton depicted by part (D2) of Fig. 1. It takes as input the sequence of pairs $\langle vars[0], vars[1]\rangle$, $\langle vars[1], vars[2]\rangle$, $\dots$, $\langle vars[n-2], vars[n-1]\rangle$ and triggers a transition for each pair. Observe that a given variable may occur in more than one pair. Each transition compares the respective values of two consecutive variables of $vars[0..n-1]$ and increments the counter c when a new inflexion is detected. The final state returns a success when the value of c is equal to $ninf$.

Synthesizing all the observations we got from these examples leads to the following remarks and definitions for a given global constraint $\mathcal{C}$:

– For a given state, no transition can be triggered indicates that the constraint $\mathcal{C}$ does not hold.
– Since all transitions starting from a given state are mutually incompatible all automata are deterministic. Let $\mathcal{M}$ denote the set of mutually incompatible conditions associated with the different transitions of an automaton.
– Let $\mathcal{S}_0, \dots, \mathcal{S}_{m-1}$ denote the sequence of subsets of variables of $\mathcal{C}$ on which the transitions are successively triggered. All these subsets contain the same number of elements and refer to some variables of $\mathcal{C}$. Since these subsets typically depend on the constraint, we leave the computation of $\mathcal{S}_0, \dots, \mathcal{S}_{m-1}$ outside the automaton. To each subset $\mathcal{S}_i$ of this sequence corresponds a variable S_i with an initial domain ranging over $[min, min + |\mathcal{M}| - 1]$, where min is a fixed integer. To each integer of this range corresponds one of the mutually incompatible conditions of $\mathcal{M}$. The sequences $S_0, \dots, S_{m-1}$ and $\mathcal{S}_0, \dots, \mathcal{S}_{m-1}$ are respectively called the *signature* and the *signature argument* of the constraint. The constraint between S_i and the variables of $\mathcal{S}_i$ is called the *signature constraint* and is denoted by $\Psi_{\mathcal{C}}(S_i, \mathcal{S}_i)$.
– From a pragmatic point the view, the task of writing a constraint checker is naturally done by writing down an imperative program where local variables, assignment

statements and control structures are used. This suggested us to consider deterministic finite automata augmented with local variables and assignment statements on these variables. Regarding control structures, we did not introduce any extra feature since the deterministic choice of which transition to trigger next seemed to be good enough.

– Many global constraints involve a variable whose value is computed from a given collection of variables. This convinced us to allow the final state of an automaton to optionally return a result. In practice, this result corresponds to the value of a local variable of the automaton in the final state.

Defining an Automaton. An automaton $\mathcal{A}$ of a constraint $\mathcal{C}$ is defined by a septuple

$$\langle \mathcal{S}ignature, \mathcal{S}ignature\mathcal{D}omain, \mathcal{S}ignature\mathcal{A}rg, \mathcal{S}ignature\mathcal{A}rg\mathcal{P}attern, \\ \mathcal{C}ounters, \mathcal{S}tates, \mathcal{T}ransitions\rangle$$

where:

– $\mathcal{S}ignature$ is the sequence of variables $S_0, \ldots, S_{m-1}$ corresponding to the signature of the constraint $\mathcal{C}$.
– $\mathcal{S}ignature\mathcal{D}omain$ is an interval which defines the range of possible values of the variables of $\mathcal{S}ignature$.
– $\mathcal{S}ignature\mathcal{A}rg$ is the signature argument $\mathcal{S}_0, \ldots, \mathcal{S}_{m-1}$ of the constraint $\mathcal{C}$. The link between the variables of $\mathcal{S}_i$ and the variable S_i $(0 \leq i < m)$ is done by writing down the signature constraint $\Psi_{\mathcal{C}}(S_i, \mathcal{S}_i)$ in such a way that arc-consistency is achieved. In our context this is done by using standard features of the CLP(FD) solver of SICStus Prolog [13] such as arithmetic constraints between two variables, propositional combinators or the global constraints programming interface.
– When used, $\mathcal{S}ignature\mathcal{A}rg\mathcal{P}attern$ defines a symbolic name for each term of $\mathcal{S}ignature\mathcal{A}rg$. These names can be used within the description of a transition for expressing an additional condition for triggering the corresponding transition.
– $\mathcal{C}ounters$ is the, possibly empty, list of all counters used in the automaton $\mathcal{A}$. Each counter is described by a term $\mathrm{t}(Counter, InitialValue, FinalVariable)$ where $Counter$ is a symbolic name representing the counter, $InitialValue$ is an integer giving the value of the counter in the initial state of $\mathcal{A}$, and $FinalVariable$ gives the variable that should be unified with the value of the counter in the final state of $\mathcal{A}$.
– $\mathcal{S}tates$ is the list of states of $\mathcal{A}$, where each state has the form $\mathrm{source}(id)$, $\mathrm{sink}(id)$ or $\mathrm{node}(id)$. id is a unique identifier associated with each state. Finally, $\mathrm{source}(id)$ and $\mathrm{sink}(id)$ respectively denote the initial and the final state of $\mathcal{A}$.
– $\mathcal{T}ransitions$ is the list of transitions of $\mathcal{A}$. Each transition t has the form $\mathrm{arc}(id_1, label, id_2)$ or $\mathrm{arc}(id_1, label, id_2, counters)$. id_1 and id_2 respectively correspond to the state just before and just after t, while $label$ depicts the value that the signature variable should have in order to trigger t. When used, $counters$ gives for each counter of $\mathcal{C}ounters$ its value after firing the corresponding transition. This value is specified by an arithmetic expression involving counters, constants, as well as usual arithmetic functions such as $+$, $-$, $\min$ or $\max$. The order used in the $counters$ list is identical to the order used in $\mathcal{C}ounters$.

Example 1. As an illustrative example we give the description of the automaton associated with the `inflexion`$(ninf, vars)$ constraint. We have:

- $\mathcal{S}ignature = S_0, S_1, \ldots, S_{n-2}$,
- $\mathcal{S}ignature\mathcal{D}omain = 0..2$,
- $\mathcal{S}ignature\mathcal{A}rg = \langle vars[0], vars[1]\rangle, \ldots, \langle vars[n-2], vars[n-1]\rangle$,
- $\mathcal{S}ignature\mathcal{A}rg\mathcal{P}attern$ is not used,
- $\mathcal{C}ounters = \mathrm{t}(c, 0, ninf)$,
- $\mathcal{S}tates = [source(s), node(i), node(j), sink(t)]$,
- $\mathcal{T}ransitions = [\mathrm{arc}(s,1,s), \mathrm{arc}(s,2,i), \mathrm{arc}(s,0,j), \mathrm{arc}(s,\$,t), \mathrm{arc}(i,1,i), \mathrm{arc}(i,2,i),$ $\mathrm{arc}(i,0,j,[c+1]), \mathrm{arc}(i,\$,t), \mathrm{arc}(j,1,j), \mathrm{arc}(j,0,j), \mathrm{arc}(j,2,i,[c+1]), \mathrm{arc}(j,\$,t)]$.

The signature constraint relating each pair of variables $\langle vars[i], vars[i+1]\rangle$ to the signature variable S_i is defined as follows: $\Psi_{\texttt{inflexion}}(S_i, vars[i], vars[i+1]) \equiv vars[i] > vars[i+1] \Leftrightarrow S_i = 0 \ \wedge vars[i] = vars[i+1] \Leftrightarrow S_i = 1 \ \wedge vars[i] < vars[i+1] \Leftrightarrow S_i = 2$. The sequence of transitions triggered on the ground instance `inflexion`$(4, [3,3,1,4,5,5,6,5,5,6,3])$ is $\frac{s}{c=0} \xrightarrow{3=3\Leftrightarrow S_0=1} s \xrightarrow{3>1\Leftrightarrow S_1=0} j \xrightarrow[c=1]{1<4\Leftrightarrow S_2=2} i \xrightarrow{4<5\Leftrightarrow S_3=2} i \xrightarrow{5=5\Leftrightarrow S_4=1} i \xrightarrow{5<6\Leftrightarrow S_5=2}$ $i \xrightarrow[c=2]{6>5\Leftrightarrow S_6=0} j \xrightarrow{5=5\Leftrightarrow S_7=1} j \xrightarrow[c=3]{5<6\Leftrightarrow S_8=2} i \xrightarrow[c=4]{6>3\Leftrightarrow S_9=0} j \xrightarrow{\$} \frac{t}{ninf=4}$. Each transition gives the corresponding condition and, eventually, the value of the counter c just after firing that transition.

3 Filtering Algorithm

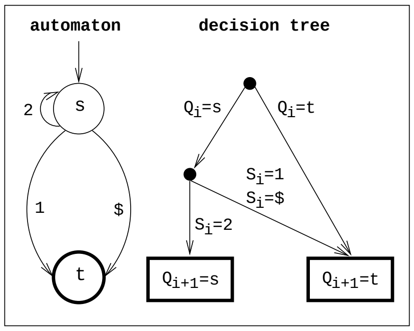
The filtering algorithm is based on the following idea. For a given global constraint $\mathcal{C}$, one can think of its automaton as a procedure that repeatedly maps a current state Q^i and counter vector $\overrightarrow{K}^i$, given a signature variable S_i, to a new state Q^{i+1} and counter vector $\overrightarrow{K}^{i+1}$, until a terminal state is reached. We then convert this procedure into a *transition constraint* $\Phi_{\mathcal{C}}(Q^i, \overrightarrow{K}^i, S_i, Q^{i+1}, \overrightarrow{K}^{i+1})$ as follows. Assuming that the automaton associated with $\mathcal{C}$ has na arcs $\mathrm{arc}(q_1, s_1, q'_1, \overrightarrow{f_1}(\overrightarrow{K})), \ldots, \mathrm{arc}(q_{na}, s_{na}, q'_{na}, \overrightarrow{f_{na}}(\overrightarrow{K}))$, the transition constraint has the following form, implemented with arithmetic, `case` [1], and `element` constraints [13]:

$$\bigvee \begin{cases} (Q^i = q_1) \wedge (S_i = s_1) \wedge (Q^{i+1} = q'_1) \wedge (\overrightarrow{K}^{i+1} = \overrightarrow{f_1}(\overrightarrow{K}^i)) \\ \vdots \\ (Q^i = q_{na}) \wedge (S_i = s_{na}) \wedge (Q^{i+1} = q'_{na}) \wedge (\overrightarrow{K}^{i+1} = \overrightarrow{f_{na}}(\overrightarrow{K}^i)) \end{cases}$$

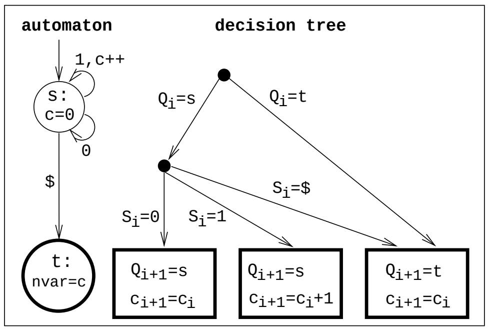
We can then arrive at a filtering algorithm for $\mathcal{C}$ by decomposing it into a conjunction of $\Phi_{\mathcal{C}}$ constraints, "threading" the state and counter variables through the conjunction. In addition to this, we need the signature constraints $\Psi_{\mathcal{C}}(S_i, \mathcal{S}_i)$ $(0 \le i < m)$ that relate each signature variables S_i to the variables of its corresponding signature argument $\mathcal{S}_i$. Filtering for the constraint $\mathcal{C}$ is provided by the conjunction of all signature and transitions constraints, (s being the start state and t being the end state):

[1] When no counter variable is used we only need a single `case` constraint to encode the disjunction expressed by the transition constraint. Since the `case` constraint [13, page 463] achieves arc-consistency, it follows that, in this context, the transition constraint achieves arc-consistency.

$$\begin{array}{ll}\Psi_{\mathcal{C}}(S_0,\mathcal{S}_0) & \wedge\ \Phi_{\mathcal{C}}(\mathrm{s},\overrightarrow{K}^0,S_0,Q^1,\overrightarrow{K}^1)\ \wedge \\ \Psi_{\mathcal{C}}(S_1,\mathcal{S}_1) & \wedge\ \Phi_{\mathcal{C}}(Q^1,\overrightarrow{K}^1,S_1,Q^2,\overrightarrow{K}^2)\ \wedge \\ \vdots & \\ \Psi_{\mathcal{C}}(S_{m-1},\mathcal{S}_{m-1}) & \wedge\ \Phi_{\mathcal{C}}(Q^{m-1},\overrightarrow{K}^{m-1},S_{m-1},Q^m,\overrightarrow{K}^m)\ \wedge \\ & \Phi_{\mathcal{C}}(Q^m,\overrightarrow{K}^m,\$,\mathrm{t},\overrightarrow{K}^{m+1})\end{array}$$



(A)



(B)

Fig. 2. Automata and decision trees for (A) $\leq_{\mathrm{lex}}$ and (B) `among`.

A couple of examples will help clarify this idea. Note that the decision tree needs to correctly handle the case when the terminal state has already been reached.

Example 2. Consider a $\overrightarrow{x} \leq_{\mathrm{lex}} \overrightarrow{y}$ constraint over vectors of length n. First, we need a signature constraint $\Psi_{\leq_{\mathrm{lex}}}$ relating each pair of arguments $x[i], y[i]$ to a signature variable S_i. This can be done as follows: $\Psi_{\leq_{\mathrm{lex}}}(S_i, x[i], y[i]) \equiv (x[i] < y[i] \Leftrightarrow S_i = 1)\ \wedge (x[i] = y[i] \Leftrightarrow S_i = 2)\ \wedge (x[i] > y[i] \Leftrightarrow S_i = 3)$. The automaton of $\leq_{\mathrm{lex}}$ and the decision tree corresponding to the transition constraint $\Phi_{\leq_{\mathrm{lex}}}$ are shown in part (A) of Fig. 2.

Example 3. Consider a `among`$(nvar, vars, values)$ constraint. First, we need a signature constraint $\Psi_{\mathtt{among}}$ relating each argument $vars[i]$ to a signature letter S_i. This can be done as follows: $\Psi_{\mathtt{among}}(S_i, vars[i], values) \equiv (vars[i] \in values \Leftrightarrow S_i = 1)\ \wedge (vars[i] \notin values \Leftrightarrow S_i = 0)$. The automaton of `among` and the decision tree corresponding to the transition constraint $\Phi_{\mathtt{among}}$ are shown in part (B) of Fig. 2.

Consistency. We consider automata where all subsets of variables in $\mathcal{S}ignature\mathcal{A}rg$ are pairwise disjoint, and that do not involve counters. Many constraints can be encoded by such automata, for instance the `global_contiguity` and `lex_lesseq` constraints presented in Fig. 1. For this kind of automata the filtering algorithm achieves arc-consistency, provided that the filtering algorithms of signature and transition constraints achieve also arc-consistency. To prove this property, consider the constraint hypergraph that represents the conjunction of all signature and transition constraints (see Fig. 3). It has two particular properties: there is no cycle in the corresponding intersection graph[2],

[2] In this graph each vertex corresponds to a constraint and there is an edge between two vertices iff the sets of variables involved in the two corresponding constraints intersect.

and for any pair of constraints the two sets of involved variables share at most one variable. Such an hypergraph is so-called *Berge-acyclic* [9]. Berge-acyclic constraint

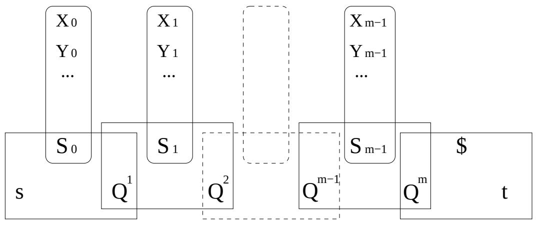


Fig. 3. Constraint hypergraph of the conjunction of transition and signature constraints in the case of disjoint $\mathcal{S}ignature\mathcal{A}rg$ sets. The i-th $\mathcal{S}ignature\mathcal{A}rg$ set $\mathcal{S}_i$ is denoted by $\{X_i, Y_i, \ldots\}$.

networks were proved to be solvable polynomially by achieving arc-consistency [21, 22]. Therefore, if all signature and transition constraints achieve arc-consistency then we obtain a complete filtering for our global constraint.

Performance. It is reasonable to ask the question whether the filtering algorithm described herein performs anywhere near the performance delivered by a hard-coded implementation of a given constraint. To this end, we have compared a version of the Balanced Incomplete Block Design problem [19, prob028] that uses a built-in $\leq_{\mathrm{lex}}$ constraint to break column symmetries with a version using our filtering based on a finite automaton for the same constraint. In a second experiment, we measured the time to find all solutions to a single $\leq_{\mathrm{lex}}$ constraint. The experiments were run in SICStus Prolog 3.11 on a 600MHz Pentium III. The results are shown in Table 1.

Table 1. Time in seconds for finding (A) the first solution of BIBD instances using built-in vs. simulated $\leq_{\mathrm{lex}}$, and (B) all solutions to a single built-in vs. simulated $\leq_{\mathrm{lex}}$ constraint.

(A)

Problem v, b, r, k, λ	**Built-in** $\leq_{\mathrm{lex}}$	**Simulated** $\leq_{\mathrm{lex}}$
$6, 50, 25, 3, 10$	0.250	0.440
$6, 60, 30, 3, 12$	0.330	0.570
$8, 14, 7, 4, 3$	0.090	0.120
$9, 120, 40, 4, 10$	1.570	2.180
$10, 90, 27, 3, 6$	1.670	2.070
$10, 120, 36, 3, 8$	3.530	3.870
$12, 88, 22, 3, 4$	1.470	2.040
$13, 104, 24, 3, 4$	1.840	2.770
$15, 70, 14, 3, 2$	1.200	1.860

(B)

$$x_i \in [0, m-1], y_i = m - i$$
$$|\overrightarrow{x}| = |\overrightarrow{y}| = m$$

m	**Built-in** $\leq_{\mathrm{lex}}$	**Simulated** $\leq_{\mathrm{lex}}$
4	0.010	0.020
5	0.110	0.170
6	1.640	2.300
7	29.530	39.100

4 Applications of this Technique

Designing Filtering Algorithm for Global Constraints. We apply this new methodology for designing filtering algorithms for the following fairly large set of global constraints. We came up with an automaton[3] for the following constraints:

- Unary constraints specifying a domain like `in` [14] or `not_in` [17].
- Channeling constraints like `domain_constraint` [28].
- Counting constraints for constraining the number of occurrences of a given set of values like `among` [6], `atleast` [17], `atmost` [17] or `count` [14].
- Sliding sequence constraints like `change` [5], `longest_change` or `smooth` [3]. `longest_change`$(size, vars, ctr)$ restricts the variable $size$ to the maximum number of consecutive variables of $vars$ for which the binary constraint ctr holds.
- Variations around the `element` constraint [20] like `element_greatereq` [25], `element_lesseq` [25] or `element_sparse` [17].
- Variations around the `maximum` constraint [4] like `max_index`$(vars, index)$. `max_index` enforces the variable $index$ to be equal to one of the positions of variables corresponding to the maximum value of the variables of $vars$.
- Constraints on words like `global_contiguity` [23], `group` [17], `group_skip_isolated_item` [3] or `pattern` [11].
- Constraints between vectors of variables like `between` [12], $\leq_{\text{lex}}$ [18], `lex_different` or `differ_from_at_least_k_pos`. Given two vectors $\vec{x}$ and $\vec{y}$ which have the same number of components, the constraints `lex_different`$(\vec{x}, \vec{y})$ and `differ_from_at_least_k_pos`$(k, \vec{x}, \vec{y})$ respectively enforce the vectors $\vec{x}$ and $\vec{y}$ to differ from at least 1 and k components.
- Constraints between n-dimensional boxes like `two_quad_are_in_contact` [17] or `two_quad_do_not_overlap` [7].
- Constraints on the shape of a sequence of variables like `inflexion` [3], `top` [8] or `valley` [8].
- Various constraints like `in_same_partition`$(var_1, var_2, partitions)$, `not_all_equal`$(vars)$ or `sliding_card_skip0`$(atleast, atmost, vars, values)$. `in_same_partition` enforces variables var_1 and var_2 to be respectively assigned to two values that both belong to a same sublist of values of $partitions$. `not_all_equal` enforces the variables of $vars$ to take more than a single value. `sliding_card_skip0` enforces that each maximum non-zero subsequence of consecutive variables of $vars$ contains at least $atleast$ and $atmost$ values from the set of values $values$.

Filtering Algorithm for a Conjunction of Global Constraints. Another typical use of our new methodology is to come up with a filtering algorithm for the conjunction of several global constraints. This is usually not easy since this implies analyzing a lot of special cases showing up from the interaction of the different considered constraints. We illustrate this point on the conjunction of the `between`$(\vec{a}, \vec{x}, \vec{b})$ [12]

[3] These automata are available in the Appendix. All signature constraints are encoded in order to achieve arc-consistency.

and the $\texttt{exactly\_one}(\overrightarrow{x}, \mathit{values})$ constraints for which we come with a filtering algorithm, which maintains arc-consistency. The $\texttt{between}$ constraint holds iff $\overrightarrow{a} \leq_{\text{lex}} \overrightarrow{x}$ and $\overrightarrow{x} \leq_{\text{lex}} \overrightarrow{b}$, while the $\texttt{exactly\_one}$ constraint holds if exactly one component of $\overrightarrow{x}$ takes its value in the set of values values.

The left-hand part of Fig. 4 depicts the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively associated with the $\texttt{between}$ and the $\texttt{exactly\_one}$ constraints, while the right-hand part gives the automaton $\mathcal{A}_3$ associated with the conjunction of these two constraints. $\mathcal{A}_3$ corresponds to the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. States of $\mathcal{A}_3$ are labeled by the two states of $\mathcal{A}_1$ and $\mathcal{A}_2$ they were issued. Transitions of $\mathcal{A}_3$ are labeled by the end symbol \$ or by a conjunction of elementary conditions, where each condition is taken in one of the following set of conditions $\{a_i < x_i, a_i = x_i, a_i > x_i\}$, $\{b_i > x_i, b_i = x_i, b_i < x_i\}$, $\{x_i \in \mathit{values}, x_i \notin \mathit{values}\}$. This makes up to $3 \cdot 3 \cdot 2 = 18$ possible combinations and leads to the signature constraint $\Psi_{\texttt{between}\wedge\texttt{exactly\_one}}(S_i, a_i, x_i, b_i, \mathit{values})$ between the signature variable S_i and the i-th component of vectors $\overrightarrow{a}$, $\overrightarrow{x}$ and $\overrightarrow{b}$:

$$S_i = \begin{cases}
0 \quad \text{if } a_i < x_i \wedge b_i > x_i \wedge x_i \notin \mathit{values}, & 9 \ \text{if } a_i < x_i \wedge b_i > x_i \wedge x_i \in \mathit{values},\\
1 \quad \text{if } a_i < x_i \wedge b_i = x_i \wedge x_i \notin \mathit{values}, & 10 \ \text{if } a_i < x_i \wedge b_i = x_i \wedge x_i \in \mathit{values},\\
2 \quad \text{if } a_i < x_i \wedge b_i < x_i \wedge x_i \notin \mathit{values}, & 11 \ \text{if } a_i < x_i \wedge b_i < x_i \wedge x_i \in \mathit{values},\\
3 \quad \text{if } a_i = x_i \wedge b_i > x_i \wedge x_i \notin \mathit{values}, & 12 \ \text{if } a_i = x_i \wedge b_i > x_i \wedge x_i \in \mathit{values},\\
4 \quad \text{if } a_i = x_i \wedge b_i = x_i \wedge x_i \notin \mathit{values}, & 13 \ \text{if } a_i = x_i \wedge b_i = x_i \wedge x_i \in \mathit{values},\\
5 \quad \text{if } a_i = x_i \wedge b_i < x_i \wedge x_i \notin \mathit{values}, & 14 \ \text{if } a_i = x_i \wedge b_i < x_i \wedge x_i \in \mathit{values},\\
6 \quad \text{if } a_i > x_i \wedge b_i > x_i \wedge x_i \notin \mathit{values}, & 15 \ \text{if } a_i > x_i \wedge b_i > x_i \wedge x_i \in \mathit{values},\\
7 \quad \text{if } a_i > x_i \wedge b_i = x_i \wedge x_i \notin \mathit{values}, & 16 \ \text{if } a_i > x_i \wedge b_i = x_i \wedge x_i \in \mathit{values},\\
8 \quad \text{if } a_i > x_i \wedge b_i < x_i \wedge x_i \notin \mathit{values}, & 17 \ \text{if } a_i > x_i \wedge b_i < x_i \wedge x_i \in \mathit{values}.
\end{cases}$$

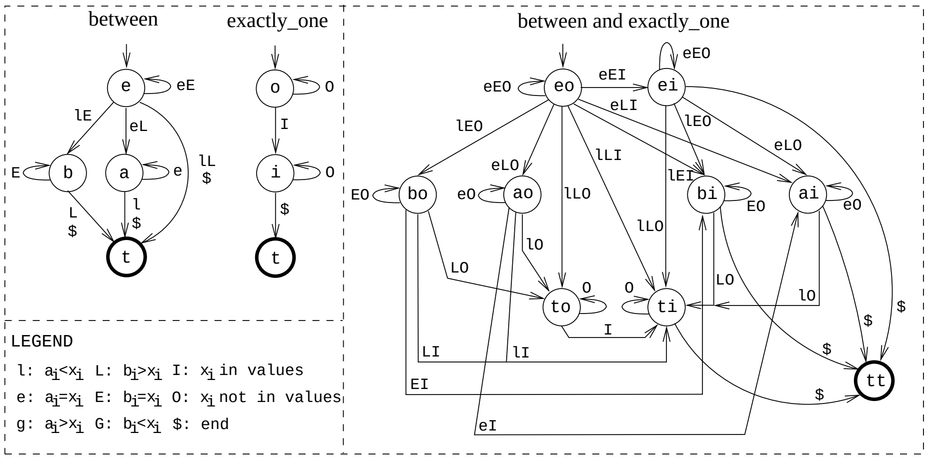


Fig. 4. Automata associated with $\texttt{between}$ and $\texttt{exactly\_one}$ and the automaton associated with their conjunction.

In order to achieve arc-consistency on the conjunction of the $\texttt{between}(\overrightarrow{a}, \overrightarrow{x}, \overrightarrow{b})$ and the $\texttt{exactly\_one}(\overrightarrow{x}, \mathit{values})$ constraints we need to have arc-consistency on

$\Psi_{\texttt{between}\wedge\texttt{exactly\_one}}(S_i, a_i, x_i, b_i, values)$. In our context this is done by using the global constraint programming facilities of SICStus Prolog [14][4]

Example 4. Consider three variables $x \in \{0,1\}$, $y \in \{0,3\}$, $z \in \{0,1,2,3\}$ subject to the conjunction of constraints $\texttt{between}(\langle 0,3,1\rangle, \langle x,y,z\rangle, \langle 1,0,2\rangle) \wedge \texttt{exactly\_one}(\langle x,y,z\rangle, \{0\})$. Even if both the `between` and the `exactly_one` constraints achieve arc-consistency, we need the automaton associated with their conjunction to find out that $z \neq 0$. This can be seen as follows: after two transitions, the automaton will be either in state `ai` or in state `bi`. However, in either state, a 0 must already have been seen, and so there is no support for $z = 0$.

Constraints Involving Finite State Systems. A third use of our methodology is for handling problems involving finite state systems where the transition diagram is known. In this context the variables of the global constraint correspond to the sequence of triggered transitions (i.e. the signature variables) and the constraint enforces ending up in a terminal state of the diagram. Let us consider the problem of the man, wolf, goat and cabbage problem to illustrate this idea.

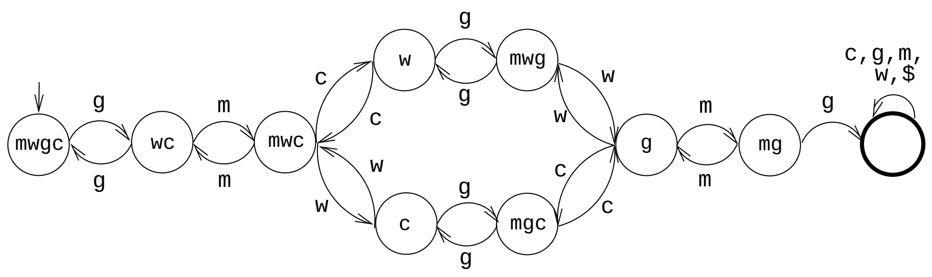


Fig. 5. Automata associated to the man, wolf, goat and cabbage problem.

Example 5. A man, a wolf, a goat and a cabbage are on the bank of a river. They want to cross to the other bank, and the man can ferry each across, one at a time. However both the wolf and the goat as well as the goat and the cabbage should not be left unattended. Fig. 5 gives the automaton associated to this problem. Each state is labeled by the subset that is on the left bank, while each transition is labeled by the action the man takes (e.g. m for cross alone, w for cross with the wolf, g for cross whith the goat, c for cross with the cabbage). On the final state, the loop models the fact that once we have reached the final state we are done.

5 Handling Relaxation for a Counter-Free Automaton

This section presents a filtering algorithm for handling constraint relaxation under the hypothesis that we don't use any counter in our automaton. It can be seen as a generalization of the algorithm used for the `regular` constraint [26].

Definition 1. *The* violation cost *of a global constraint is the minimum number of subsets of its signature argument for which it is necessary to change at least one variable in order to get back to a solution.*

[4] The corresponding code is available in the Appendix..

When these subsets form a partition over the variables of the constraint and when they consist of a single element, this cost is in fact the minimum number of variables to unassign in order to get back to a solution. As in [27], we add a cost variable $cost$ as an extra argument of the constraint. Our filtering algorithm first evaluates the minimum cost value $\mathcal{M}in$. Then, according to $\max(cost)$, it prunes values that cannot belong to a solution.

Example 6. Consider the constraint `global_contiguity`$([V_0, V_1, V_2, V_3, V_4, V_5, V_6])$ with the following current domains for variables V_i: $[\{0,1\}, \{1\}, \{1\}, \{0\}, \{1\}, \{0,1\}, \{1\}]$. The constraint is violated because there are necessarily at least two distinct sequences of consecutive 1. To get back to a state that can lead to a solution, it is enough to turn the fourth value to 1. One can deduce $\mathcal{M}in = 1$. Consider now the relaxed form `soft_global_contiguity`$([V_0, V_1, V_2, V_3, V_4, V_5, V_6], cost)$ and assume $\max(cost) = 1$. The filtering algorithm should remove value 0 from V_5. Indeed, selecting value 0 for variable V_5 entails a minimum violation cost of 2. Observe that for this constraint the signature variables $S_0, S_1, S_2, S_3, S_4, S_5, S_6$ are $V_0, V_1, V_2, V_3, V_4, V_5, V_6$.

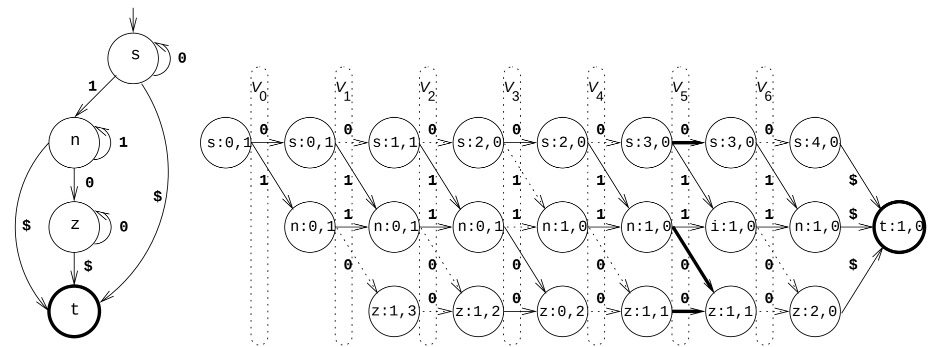
As in the algorithm of Pesant [26], our consistency algorithm builds a layered acyclic directed multigraph $\mathcal{G}$. Each layer of $\mathcal{G}$ contains a different node for each state of our automaton. Arcs only appear between consecutive layers. Given two nodes n_1 and n_2 of two consecutive layers, q_1 and q_2 denote their respective associated state. There is an arc a from n_1 to n_2 iff, in the automaton, there is an arc arc(q_1, v, q_2) from q_1 to q_2. The arc a is labeled with the value v. Arcs corresponding to transitions that cannot be triggered according to the current domain of the signature variables $S_0, \ldots, S_{m-1}$ are marked as *infeasible*. All other arcs are marked as *feasible*. Finally, we discard isolated nodes from our layered multigraph. Since our automaton has a single initial state and a single final state, $\mathcal{G}$ has one source and one sink respectively denoted by $source$ and $sink$.

Example 6 continued. Part (A) of Fig. 6 recalls the automaton of the `global_contiguity` constraint, while part (B) gives the multigraph $\mathcal{G}$ associated with the `soft_global_contiguity` constraint previously introduced. Each arc is labeled by the condition associated to its corresponding transition. Each node contains the name of the corresponding automaton state. Numbers in a node will be explained later on. Infeasible arcs are represented with a dotted line.

We now explain how to use the multigraph $\mathcal{G}$ to evaluate the minimum violation cost $\mathcal{M}in$ and to prune the signature variables according to the maximum allowed violation cost $\max(cost)$.

Evaluating the minimum violation cost $\mathcal{M}in$ can be seen as finding the path from the source to the sink of $\mathcal{G}$ that contains the smallest number of infeasible arcs. This can be done by performing a topological sort starting from the source of $\mathcal{G}$. While performing the topological sort, we compute for each node n_k of $\mathcal{G}$ the minimum number of infeasible arcs from the source of $\mathcal{G}$ to n_k. This number is recorded in $before[n_k]$. At the end of the topological sort, the minimum violation cost $\mathcal{M}in$ we search for, is equal to $before[sink]$.

Notation 1 *Let i be assignable to a signature variable S_l. $\mathcal{M}in_l^i$ denotes the minimum violation cost value according to the hypothesis that we assign i to S_l.*



(A) Automaton for global_contiguity

(B) Graph of potential executions of the automaton of global_contiguity according to $V_0, V_1, V_2, V_3, V_4, V_5, V_6$

Fig. 6. Relaxing the `global_contiguity` constraint.

To prune domains of signature variables we need to compute the quantity $\mathcal{M}in_l^i$. In order to do so, we introduce the quantity $\mathit{after}[n_k]$ for a node n_k of $\mathcal{G}$: $\mathit{after}[n_k]$ is the minimum number of infeasible arcs on all paths from n_k to sink. It is computed by performing a second topological sort starting from the sink of $\mathcal{G}$. Let $\mathcal{A}_l^i$ denote the set of arcs of $\mathcal{G}$, labeled by i, for which the origin has a rank of l. The quantity $\min\limits_{a \to b \in \mathcal{A}_l^i} (\mathit{before}[a] + \mathit{after}[b])$ represents the minimum violation cost under the hypothesis that S_l remains assigned to i. If that quantity is greater than $\mathcal{M}in$ then there is no path from source to sink which uses an arc of $\mathcal{A}_l^i$ and which has a number of infeasible arcs equal to $\mathcal{M}in$. In that case the smallest cost we can achieve is $\mathcal{M}in + 1$. Therefore we have:

$$\mathcal{M}in_l^i = \min(\min_{a \to b \in \mathcal{A}_l^i} (\mathit{before}[a] + \mathit{after}[b]), \mathcal{M}in + 1)$$

The filtering algorithm is then based on the following theorem:

Theorem 1. *Let i be a value from the domain of a signature variable S_l. If $\mathcal{M}in_l^i > \max(\mathit{cost})$ then i can be removed from S_l.*

The cost of the filtering algorithm is dominated by the two topological sorts. They have a cost proportional to the number of arcs of $\mathcal{G}$ which is bounded by the number of signature variables times the number of arcs of the automaton.

Example 6 continued. Let us come back to the instance of Fig. 6. Beside the state's name, each node n_k of part (B) of Fig. 6 gives the values of $\mathit{before}[n_k]$ and of $\mathit{after}[n_k]$. Since $\mathit{before}[\mathit{sink}] = 1$ we have that the minimum cost violation is equal to 1. Pruning can be potentially done only for signature variables having more than one value. In our example this corresponds to variables V_0 and V_5. So we evaluate the four quantities $\mathcal{M}in_0^0 = \min(0+1, 2) = 1$, $\mathcal{M}in_0^1 = \min(0+1, 2) = 1$, $\mathcal{M}in_5^0 = \min(\min(3+0, 1+1, 1+1), 2) = 2$, $\mathcal{M}in_5^1 = \min(\min(3+0, 1+0), 2) = 1$. If $\max(\mathit{cost})$ is equal to 1 we can remove value 0 from V_5. The corresponding arcs are depicted with a thick line in Fig. 6.

6 Conclusion and Perspectives

The automaton description introduced in this reportcan be seen as a restricted programming language. This language is used for writing down a constraint checker, which verifies whether a ground instance of a constraint is satisfied or not. This checker allows pruning the variables of a non-ground instance of a constraint by simulating all potential executions of the corresponding program according to the current domain of the variables of the relevant constraint. This simulation is achieved by encoding all potential executions of the automaton as a conjunction of signature and transition constraints and by letting the usual constraint propagation deducing all the relevant information. We want to stress the key points of this approach:

– Within the context of global constraints, it was implicitly assumed that providing a constraint checker is a much easier task than coming up with a filtering algorithm. It was also commonly admitted that the design of filtering algorithms is a difficult task which involves creativity and which cannot be automatized. We have shown that this is not the case any more if one can afford to provide a constraint checker.
– Non-determinism has played a key role by augmenting programming languages with backtracking facilities [16], which was the origin of logic programming. Non-determinism also has a key role to play in the systematic design of filtering algorithms: finding a filtering algorithm can be seen as the task of executing in a non-deterministic way the deterministic program corresponding to a constraint checker and to extract the relevant information which for sure occurs under any circumstances. This can indeed be achieved by using constraint programming.

We finally present different perspectives of this work.

– A natural continuation would be to extend the automaton description in order to get closer to a classical imperative programming language. This would allow reusing directly available checkers in order to systematically get a filtering algorithm.
– Another interesting question is about identifying other structural conditions on the signature and transition constraints that can guarantee arc-consistency for the original global constraint.

Acknowledgements

We are grateful to Irit Katriel for suggesting us the use of topological sort for the relaxation part, and to Christian Bessière for his helpful comments with respect to Berge-acyclic CSP's.

References

1. A.G.Frutos, Q.Liu, A.J.Thiel, A.M.W.Sanner, A.E.Condon, L.M.Smith, and R.M.Corn. Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research*, 25:4748–4757, 1997.

2. J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artificial Intelligence*, 135:199–234, 2002.
3. N. Beldiceanu. Global constraints as graph properties on structured network of elementary constaints of the same type. In R. Dechter, editor, *CP'2000, Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 52–66. Springer-Verlag, 2000.
4. N. Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In T. Walsh, editor, *CP'2001, Int. Conf. on Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 211–224. Springer-Verlag, 2001.
5. N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. In P. Codognet, editor, *ICLP'2001, Int. Conf. on Logic Programming*, volume 2237 of *LNCS*, pages 59–73. Springer-Verlag, 2001.
6. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
7. N. Beldiceanu, Q. Guo, and S. Thiel. Non-overlapping constraints between convex polytopes. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 392–407. Springer-Verlag, 2001.
8. N. Beldiceanu and E. Poder. Cumulated profiles of minimum and maximum resource utilisation. In *Ninth Int. Conf. on Project Management and Scheduling*, 2004.
9. C. Berge. Graphs and hypergraphs. *Dunod, Paris*, 1970.
10. B. Boigelot and P. Wolper. Representing arithmetic constraints with finite automata: An overview. In Peter J. Stuckey, editor, *ICLP'2002, Int. Conf. on Logic Programming*, volume 2401 of *LNCS*, pages 1–19. Springer-Verlag, 2002.
11. S. Bourdais, P. Galinier, and G. Pesant. HIBISCUS: A constraint programming application to staff scheduling in health care. In F. Rossi, editor, *CP'2003, Principles and Practice of Constraint Programming*, volume 2833 of *LNCS*, pages 153–167. Springer-Verlag, 2003.
12. M. Carlsson and N. Beldiceanu. From constraints to finite automata to filtering algorithms. In D. Schmidt, editor, *Proc. ESOP2004*, volume 2986 of *LNCS*, pages 94–108. Springer-Verlag, 2004.
13. M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 3.11 edition, January 2004. `http://www.sics.se/sicstus/`.
14. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.
15. C.Flamm, I.L.Hofacker, and P.F.Stadler. RNA in silico: The computational biology of RNA secondary structures. *Adv. Complex Syst.*, 2:5–90, 1999.
16. J. Cohen. Non-deterministic algorithms. *ACM Computing Surveys*, 11(2):79–94, 1979.
17. COSYTEC. *CHIP Reference Manual*, v5 edition, 2003.
18. A. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming (CP'2002)*, volume 2470 of *LNCS*, pages 93–108. Springer-Verlag, 2002.
19. I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical Report APES-09-1999, APES, 1999. `http://www.csplib.org`.
20. P. Van Hentenryck and J.-P. Carillon. Generality vs. specificity: an experience with AI and OR techniques. In *National Conference on Artificial Intelligence (AAAI-88)*, 1988.
21. P. Janssen and M-C. Vilarem. Problèmes de satisfaction de contraintes: techniques de résolution et application à la synthèse de peptides. *Research Report C.R.I.M.*, 54, 1988.
22. P. Jégou. Contribution à l'étude des problèmes de satisfaction de contraintes: algorithmes de propagation et de résolution. Propagation de contraintes dans les réseaux dynamiques. *PhD Thesis*, 1991.
23. M. Maher. Analysis of a global contiguity constraint. In *Workshop on Symmetry on Rule-Based Constraint Reasoning and Programming*, 2002. held along CP-2002.

24. M. Milano. *Constraint and integer programming*. Kluwer Academic Publishers, 2004. ISBN 1-4020-7583-9.
25. G. Ottosson, E. Thorsteinsson, and J. N. Hooker. Mixed global constraints and inference in hybrid IP-CLP solvers. In *CP'99 Post-Conference Workshop on Large-Scale Combinatorial Optimization and Constraints*, pages 57–78, 1999.
26. G. Pesant. A regular language membership constraint for sequence of variables. In *Workshop on Modelling and Reformulation Constraint Satisfaction Problems*, pages 110–119, 2003.
27. T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 451–463. Springer-Verlag, 2001.
28. P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Principles and Practice of Constraint Programming (CP'2000)*, volume 1894 of *LNCS*, pages 369–383. Springer-Verlag, 2000.
29. N. R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *National Conference on Artificial Intelligence (AAAI-92)*, pages 453–458. AAAI Press, 1992.

A List of Constraints and their Corresponding Automata

A.1 among(NVAR,VARIABLES,VALUES) [6]

NVAR is the number of variables of the collection VARIABLES which take their value in VALUES.

```
?- among(3,
         [[var-4],[var-5],[var-5],[var-4],[var-1]],
         [[val-1],[val-5],[val-8]]).
true.

% 0: not_in(VAR,VALUES)
% 1: in(VAR,VALUES)
among(NVAR, VARIABLES, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        among_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s,[C+1]),
                   arc(s,$,t      )],
                  [C],[0],[NVAR]).

among_signature([], [], _).
among_signature([[var-VAR]|VARs], [S|Ss], SET_OF_VALUES) :-
        VAR in_set SET_OF_VALUES #<=> S,
        among_signature(VARs, Ss, SET_OF_VALUES).
```

A.2 atleast(N,VARIABLES,VALUE) [17]

At least N variables of the VARIABLES collection are assigned to value VALUE.

```
?- atleast(2,[[var-4],[var-2],[var-4],[var-5]],4)).
true.

atleast(N, VARIABLES, VALUE) :-
        N #=< M,
        among(M, VARIABLES, [[val-VALUE]]).
```

A.3 atmost(N,VARIABLES,VALUE) [17]

At most N variables of the VARIABLES collection are assigned to value VALUE.

```
?- atmost(1,[[var-4],[var-2],[var-4],[var-5]],2)
true.

atmost(N, VARIABLES, VALUE) :-
        N #>= M,
        among(M, VARIABLES, [[val-VALUE]]).
```

A.4 between(As,Xs,Bs) [12]

Vector X is lexicographically greater than or equal to vector A and vector X is lexicographically less than or equal to vector B.

```
?- between([[var-4],[var-2]], [[var-4],[var-2]], [[var-4],[var-3]]).
true.

% 0: A#<X #/\ X#<B
% 1: A#<X #/\ X#=B
% 2: A#<X #/\ X#>B
% 3: A#=X #/\ X#<B
% 4: A#=X #/\ X#=B
% 5: A#=X #/\ X#>B
% 6: A#>X #/\ X#<B
% 7: A#>X #/\ X#=B
% 8: other
between(As, Xs, Bs) :-
        between_signature(As, Xs, Bs, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..8,
                  [source(s),node(a),node(b),sink(t)],
                  [arc(s,4,s),
                   arc(s,0,t),
                   arc(s,$,t),
                   arc(s,3,a),
                   arc(s,1,b),
                   arc(a,3,a),
                   arc(a,4,a),
                   arc(a,5,a),
                   arc(a,0,t),
                   arc(a,1,t),
                   arc(a,2,t),
                   arc(a,$,t),
                   arc(b,1,b),
                   arc(b,4,b),
                   arc(b,7,b),
                   arc(b,0,t),
                   arc(b,3,t),
                   arc(b,6,t),
                   arc(b,$,t)],
                  [], [], []).

between_signature([], [], [], []).
between_signature([[var-A1]|As], [[var-X1]|Xs], [[var-B1]|Bs], [L1|Ls]) :-
        Adown is A1-1,
        Aup   is A1+1,
        Bdown is B1-1,
        Bup   is B1+1,
        filter_cases([(A1+1<B1 -> (Aup..Bdown)-0),
                      (A1<B1 -> (B1..B1)-1),
                      (true  -> (Bup..sup)-2),
                      (A1<B1 -> (A1..A1)-3),
                      (A1=:=B1 -> (A1..A1)-4),
                      (A1>B1 -> (A1..A1)-5),
                      (true  -> (inf..Adown)-6),
                      (A1>B1 -> (B1..B1)-7),
                      (A1>B1+1 -> (Bup..Adown)-8)],
                     Cases),
        leaf_nodes(Cases, Leaves, L),
        case(X-L, [X1-L1], [node(-1,X,Cases)|Leaves]),
        between_signature(As, Xs, Bs, Ls).

filter_cases([], []).
filter_cases([(Cond->Case)|L1], [Case|L2]) :-
        call(Cond), !,
        filter_cases(L1, L2).
filter_cases([_|L1], L2) :-
        filter_cases(L1, L2).

leaf_nodes([], [], _).
leaf_nodes([_-N|Cases], [node(N,L,[N..N])|Nodes], L) :-
        leaf_nodes(Cases, Nodes, L).
```

A.5 between_exactly_one(As,Xs,Bs,VALUES)

Vector Xs is lexicographically greater than or equal to vector As and vector Xs is lexicographically less than or equal to vector Bs and exactly one component of vector Xs has a value in the set of values VALUES.

```
?- between_exactly_one([[var-4],[var-2]], [[var-4],[var-2]], [[var-4],[var-3]],
                       [[val-1],[val-2],[val-6]]).
true.

% signature: Lb+La where
% Lb =              0 - (A#<X #/\ X#<B)
%                   1 - (A#<X #/\ X#=B)
%                   2 - (A#<X #/\ X#>B)
%                   3 - (A#=X #/\ X#<B)
%                   4 - (A#=X #/\ X#=B)
%                   5 - (A#=X #/\ X#>B)
%                   6 - (A#>X #/\ X#<B)
%                   7 - (A#>X #/\ X#=B)
%                   8 - (A#>X #/\ X#>B)
% La =              0 - X not in VALUES
%                   9 - X in VALUES
between_exactly_one(As, Xs, Bs, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, Set),
        Am in_set Set,
        between_exactly_one_signature(As, Xs, Bs, Ss, Am),
        automaton(Ss, _,
                  Ss, 0..17,
                  [source(eo),
                   node(ei),
                   node(ao),
                   node(ai),
                   node(bo),
                   node(bi),
                   node(to),
                   node(ti),
                   sink(tt)],
                  [arc(eo, 4,eo), % eEO
                   arc(eo, 3,ao), % eLO
                   arc(eo, 1,bo), % lEO
                   arc(eo, 0,to), % lLO
                   arc(eo,13,ei), % eEI
                   arc(eo,12,ai), % eLI
                   arc(eo,10,bi), % lEI
                   arc(eo, 9,ti), % lLI
                   %
                   arc(ei,4,ei), % eEO
                   arc(ei,3,ai), % eLO
                   arc(ei,1,bi), % lEO
                   arc(ei,0,ti), % lLO
                   arc(ei,$,tt), % $
                   %
                   arc(ao, 3,ao), % eLO
                   arc(ao, 4,ao), % eEO
                   arc(ao, 5,ao), % eGO
                   arc(ao, 0,to), % lLO
                   arc(ao, 1,to), % lEO
                   arc(ao, 2,to), % lGO
                   arc(ao,12,ai), % eLI
                   arc(ao,13,ai), % eEI
                   arc(ao,14,ai), % eGI
                   arc(ao, 9,ti), % lLI
                   arc(ao,10,ti), % lEI
                   arc(ao,11,ti), % lGI
                   %
                   arc(ai,3,ai), % eLO
                   arc(ai,4,ai), % eEO
                   arc(ai,5,ai), % eGO
                   arc(ai,0,ti), % lLO
                   arc(ai,1,ti), % lEO
                   arc(ai,2,ti), % lGO
                   arc(ai,$,tt), % $
                   %
                   arc(bo, 1,bo), % lEO
                   arc(bo, 4,bo), % eEO
                   arc(bo, 7,bo), % gEO
                   arc(bo, 0,to), % lLO
                   arc(bo, 3,to), % eLO
                   arc(bo, 6,to), % gLO
                   arc(bo,10,bi), % lEI
                   arc(bo,13,bi), % eEI
                   arc(bo,16,bi), % gEI
                   arc(bo, 9,ti), % lLI
                   arc(bo,12,ti), % eLI
                   arc(bo,15,ti), % gLI
                   %
                   arc(bi,1,bi), % lEO
                   arc(bi,4,bi), % eEO
                   arc(bi,7,bi), % gEO
                   arc(bi,0,ti), % lLO
                   arc(bi,3,ti), % eLO
                   arc(bi,6,ti), % gLO
                   arc(bi,$,tt), % $
                   %
                   arc(to, 0,to),
                   arc(to, 1,to),
                   arc(to, 2,to),
                   arc(to, 3,to),
                   arc(to, 4,to),
                   arc(to, 5,to),
                   arc(to, 6,to),
                   arc(to, 7,to),
                   arc(to, 8,to),
                   arc(to, 9,ti),
                   arc(to,10,ti),
                   arc(to,11,ti),
                   arc(to,12,ti),
                   arc(to,13,ti),
                   arc(to,14,ti),
                   arc(to,15,ti),
                   arc(to,16,ti),
                   arc(to,17,ti),
                   %
                   arc(ti,0,ti),
                   arc(ti,1,ti),
                   arc(ti,2,ti),
                   arc(ti,3,ti),
                   arc(ti,4,ti),
                   arc(ti,5,ti),
                   arc(ti,6,ti),
                   arc(ti,7,ti),
                   arc(ti,8,ti),
                   arc(ti,$,tt)],
                  [], [], []).

between_exactly_one_signature([], [], [], [], _).
between_exactly_one_signature([[var-A1]|As], [[var-X1]|Xs], [[var-B1]|Bs], [L1|Ls], Am) :-
        fd_global(btw_exactly_one_sig(A1,X1,B1,Am,L1), [], [dom(X1),dom(L1)]),
        between_exactly_one_signature(As,Xs,Bs,Ls,Am).

:- multifile clpfd:dispatch_global/4.

clpfd:dispatch_global(btw_exactly_one_sig(A,X,B,Am,Sig), [], [], [X in_set Xset,Sig in_set Sset]) :-
        fd_set(Am, Among),
        findall(X-Sig, (indomain(X), btw_exactly_one_sig1(A,X,B,Among,Sig)), L1),
        keys_and_values(L1, Xs, Ss1),
        list_to_fdset(Xs, Xset),
        sort(Ss1, Ss2),
        list_to_fdset(Ss2, Sset).

btw_exactly_one_sig1(A,X,B,Among,Sig) :-
        (X in_set Among -> Term=9 ; Term=0),
        (   A < X, X < B -> Sig is 0 + Term
        ;   A < X, X=:=B -> Sig is 1 + Term
        ;   A < X, X > B -> Sig is 2 + Term
        ;   A=:=X, X < B -> Sig is 3 + Term
        ;   A=:=X, X=:=B -> Sig is 4 + Term
        ;   A=:=X, X > B -> Sig is 5 + Term
        ;   A > X, X < B -> Sig is 6 + Term
        ;   A > X, X=:=B -> Sig is 7 + Term
        ;   A > X, X > B -> Sig is 8 + Term
        ).

keys_and_values([], [], []).
keys_and_values([Key-Value|Pairs], [Key|Keys], [Value|Values]) :-
        keys_and_values(Pairs, Keys, Values).
```

A.6 `change(NCHANGE,VARIABLES,CTR)` [17, 5]

`NCHANGE` is the number of times that constraint `CTR` holds on consecutive variables of the collection `VARIABLES`.

```
?- change(3,[[var-4],[var-4],[var-3],[var-4],[var-1]],=\=).
true.

% CTR: =
% 0: VAR1=\=VAR2
% 1: VAR1=VAR2
%
% CTR: =\=
% 0: VAR1=VAR2
% 1: VAR1=\=VAR2
%
% CTR: <
% 0: VAR1>=VAR2
% 1: VAR1<VAR2
%
% CTR: >=
% 0: VAR1<VAR2
% 1: VAR1>=VAR2
%
% CTR: >
% 0: VAR1=<VAR2
% 1: VAR1>VAR2
%
% CTR: =<
% 0: VAR1>VAR2
% 1: VAR1=<VAR2
change(NCHANGE, VARIABLES, CTR) :-
        change_signature(VARIABLES, SIGNATURE, CTR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s,[C+1]),
                   arc(s,$,t      )],
                  [C],[0],[NCHANGE]).

change_signature([], [], _).
change_signature([_], [], _) :- !.
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =) :- !,
        VAR1 #= VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, =).
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =\=) :- !,
        VAR1 #\= VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, =\=).
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], <) :- !,
        VAR1 #< VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, <).
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >=) :- !,
        VAR1 #>= VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, >=).
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >) :- !,
        VAR1 #> VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, >).
change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =<) :- !,
        VAR1 #=< VAR2 #<=> S,
        change_signature([[var-VAR2]|VARs], Ss, =<).
```

A.7 circular_change(NCHANGE,VARIABLES,CTR)

NCHANGE is the number of times that CTR holds on consecutive variables of the collection VARIABLES. The last and the first variables of the collection VARIABLES are also considered to be consecutive.

```
?- circular_change(4,[[var-4],[var-4],[var-3],[var-4],[var-1]],=\=).
true.

% CTR: =
% 0: VAR1=\=VAR2
% 1: VAR1=VAR2
%
% CTR: =\=
% 0: VAR1=VAR2
% 1: VAR1=\=VAR2
%
% CTR: <
% 0: VAR1>=VAR2
% 1: VAR1<VAR2
%
% CTR: >=
% 0: VAR1<VAR2
% 1: VAR1>=VAR2
%
% CTR: >
% 0: VAR1=<VAR2
% 1: VAR1>VAR2
%
% CTR: =<
% 0: VAR1>VAR2
% 1: VAR1=<VAR2
circular_change(NCHANGE, VARIABLES, CTR) :-
        VARIABLES = [V1|_],
        append(VARIABLES, [V1], CVARIABLES),
        circular_change_signature(CVARIABLES, SIGNATURE, CTR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s,[C+1]),
                   arc(s,$,t      )],
                  [C],[0],[NCHANGE]).

circular_change_signature([], [], _).
circular_change_signature([_], [], _) :- !.
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =) :- !,
        VAR1 #= VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, =).
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =\=) :- !,
        VAR1 #\= VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, =\=).
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], <) :- !,
        VAR1 #< VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, <).
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >=) :- !,
        VAR1 #>= VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, >=).
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >) :- !,
        VAR1 #> VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, >).
circular_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =<) :- !,
        VAR1 #=< VAR2 #<=> S,
        circular_change_signature([[var-VAR2]|VARs], Ss, =<).
```

A.8 `count_(VALUE,VARIABLES,RELOP,NVAR)` [13]

Let `N` be the number of variables of the `VARIABLES` collection assigned to value `VAL`; Enforce condition `N` `RELOP` `NVAR` to hold.

```
?- count_(5,[[var-4],[var-5],[var-5],[var-4],[var-5]],>=,2).
true.

% 0: VAR=\=VALUE
% 1: VAR=VALUE
count_(VALUE, VARIABLES, RELOP, NVAR) :-
        length(VARIABLES, N),
        NIN in 0..N,
        count_signature(VARIABLES, SIGNATURE, VALUE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s,[C+1]),
                   arc(s,$,t      )],
                  [C],[0],[NIN]),
        count_relop(RELOP, NIN, NVAR).

count_signature([], [], _).
count_signature([[var-VAR]|VARs], [S|Ss], VALUE) :-
        VAR #= VALUE #<=> S,
        count_signature(VARs, Ss, VALUE).

count_relop(=  , NIN, NVAR) :- NIN #=  NVAR.
count_relop(=\=, NIN, NVAR) :- NIN #\= NVAR.
count_relop(<  , NIN, NVAR) :- NIN #<  NVAR.
count_relop(>= , NIN, NVAR) :- NIN #>= NVAR.
count_relop(>  , NIN, NVAR) :- NIN #>  NVAR.
count_relop(=< , NIN, NVAR) :- NIN #=< NVAR.
```

A.9 `counts(VALUES,VARIABLES,RELOP,LIMIT)`

Let `N` be the number of variables of the `VARIABLES` collection assigned to a value of the `VALUES` collection. Enforce condition `N` `RELOP` `LIMIT` to hold. Inspired by `count`.

```
?- counts([[val-1],[val-3],[val-4],[val-9]],
          [[var-4],[var-5],[var-5],[var-4],[var-1],[var-5]],=,3).
true.

% 0: not_in(VAR,VALUES)
% 1: in(VAR,VALUES)
counts(VALUES, VARIABLES, RELOP, LIMIT) :-
```

```
        length(VARIABLES, N),
        NIN in 0..N,
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        counts_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s,[C+1]),
                   arc(s,$,t      )],
                  [C],[0],[NIN]),
        count_relop(RELOP, NIN, LIMIT).

counts_signature([], [], _).
counts_signature([[var-VAR]|VARs], [S|Ss], SET_OF_VALUES) :-
        VAR in_set SET_OF_VALUES #<=> S,
        counts_signature(VARs, Ss, SET_OF_VALUES).
```

A.10 deepest_valley(DEPTH,VARIABLES)

DEPTH is the maximum value of the quantity $\min(V_i, V_k) - V_j$ where V_i, V_j, V_k are three variables of the VARIABLES collection such that $i < j < k$, $V_i > V_j$ and $V_j < V_k$. If no such variables exists DEPTH is equal to 0.

```
?- deepest_valley(3,[[var-5],[var-6],[var-6],[var-3],[var-5],
                     [var-1],[var-1],[var-4],[var-1]]).
true.

% 0: VAR1<VAR2
% 1: VAR1=VAR2
% 2: VAR1>VAR2
deepest_valley(DEPTH, VARIABLES) :-
        deepest_valley_signature(VARIABLES, SIGNATURE, PAIRS),
        automaton(PAIRS, VAR1-VAR2, SIGNATURE, 0..2,
                  [source(s),node(u),sink(t)],
                  [arc(s,0,s                                 ),
                   arc(s,1,s                                 ),
                   arc(s,2,u,[C,VAR1]                        ),
                   arc(s,$,t                                 ),
                   arc(u,0,s,[max(C,min(H-VAR1,VAR2-VAR1)),H]),
                   arc(u,1,u                                 ),
                   arc(u,2,u,[C,VAR1]                        ),
                   arc(u,$,t                                 )],
                  [C,H],[0,0],[DEPTH,_]).

deepest_valley_signature([], [], []).
deepest_valley_signature([_], [], []).
deepest_valley_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], [VAR1-VAR2|PAIRS]) :-
        VAR1 #< VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #> VAR2 #<=> S #= 2,
        deepest_valley_signature([[var-VAR2]|VARs], Ss, PAIRS).
```

A.11 differ_from_at_least_k_pos(K,VECTOR1,VECTOR2)

Enforce two vectors VECTOR1 and VECTOR2 to differ from at least K positions. Inspired by [1].

```
?- differ_from_at_least_k_pos(3,[[var-2],[var-5],[var-2],[var-0]],
                                [[var-3],[var-6],[var-2],[var-1]]).
true.

% 0: X=\=Y
% 1: X=Y
differ_from_at_least_k_pos(K, VECTOR1, VECTOR2) :-
        differ_from_at_least_k_pos_signature(VECTOR1, VECTOR2, SIGNATURE),
        Sum #>= K,
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s,[C+1]),
                   arc(s,1,s      ),
                   arc(s,$,t      )],
                  [C],[0],[Sum]).

differ_from_at_least_k_pos_signature([], [], []).
differ_from_at_least_k_pos_signature([[var-VAR1]|VARS1], [[var-VAR2]|VARS2], [S|Ss]) :-
        VAR1 #= VAR2 #<=> S,
        differ_from_at_least_k_pos_signature(VARS1, VARS2, Ss).
```

A.12 domain_constraint(VAR,VALUES) [28]

Make the link between a domain variable VAR and those 0-1 variables that are associated to each potential value of VAR: The 0-1 variable associated to the value taken by variable VAR is equal to 1, while the remaining 0-1 variables are all equal to 0.

```
?- domain_constraint(5, [[var01-0, value-9], [var01-1, value-5],
                         [var01-0, value-2], [var01-0, value-7]]).
true.

% 0: VAR=VALUE is not equivalent to VAR01=1
% 1: VAR=VALUE is     equivalent to VAR01=1
domain_constraint(VAR, VALUES) :-
        domain_constraint_signature(VALUES, SIGNATURE, VAR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s),
                   arc(s,$,t)],
                  [],[],[]).

domain_constraint_signature([], [], _).
domain_constraint_signature([[var01-VAR01,value-VALUE]|VALUES], [S|Ss], VAR) :-
        ((VAR#=VALUE) #<=> VAR01) #<=> S,
        domain_constraint_signature(VALUES, Ss, VAR).
```

A.13 elem(ITEM,TABLE)

ITEM is equal to one of the entries of the table TABLE.

```
?- elem([[index-3, value-2]],
        [[index-1, value-6],[index-2, value-9],[index-3, value-2],[index-4, value-9]]).
true.

% 0: ITEM_INDEX=\=TABLE_INDEX or  ITEM_VALUE=\=TABLE_VALUE
% 1: ITEM_INDEX = TABLE_INDEX and ITEM_VALUE = TABLE_VALUE
elem(ITEM, TABLE) :-
        ITEM = [[index-ITEM_INDEX, value-ITEM_VALUE]],
        elem_signature(TABLE, SIGNATURE, ITEM_INDEX, ITEM_VALUE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

elem_signature([], [], _, _).
elem_signature([[index-TABLE_INDEX,value-TABLE_VALUE]|TABLEs], [S|Ss],
               ITEM_INDEX, ITEM_VALUE) :-
        ((ITEM_INDEX #= TABLE_INDEX) #/\ (ITEM_VALUE #= TABLE_VALUE)) #<=> S,
        elem_signature(TABLEs, Ss, ITEM_INDEX, ITEM_VALUE).
```

A.14 `element_(INDEX,TABLE,VALUE)` [20]

`VALUE` is equal to the `INDEX`-th item of `TABLE`.

```
?- element_(3, [[value-6],[value-9],[value-2],[value-9]], 2).
true.

% 0: INDEX=\=TABLE_KEY or  VALUE=\=TABLE_VALUE
% 1: INDEX = TABLE_KEY and VALUE = TABLE_VALUE
element_(INDEX, TABLE, VALUE) :-
        length(TABLE, N),
        INDEX in 1..N,
        element_signature(TABLE, INDEX, VALUE, 1, SIGNATURE),
        automaton(TABLE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

element_signature([], _, _, _, []).
element_signature([[value-TABLE_VALUE]|Ts], INDEX, VALUE, TABLE_KEY, [B|Bs]) :-
        INDEX#=TABLE_KEY #/\ VALUE#=TABLE_VALUE #<=> B,
        TABLE_KEY1 is TABLE_KEY+1,
        element_signature(Ts, INDEX, VALUE, TABLE_KEY1, Bs).
```

A.15 `element_greatereq(ITEM,TABLE)` [25]

`ITEM.value` is greater than or equal to one of the entries (i.e. the `value` attribute) of the table `TABLE`.

```
?- element_greatereq([[index-1, value-8]],
                     [[index-1, value-6],[index-2, value-9],
                      [index-3, value-2],[index-4, value-9]]).
true.

% 0: ITEM_INDEX=\=TABLE_INDEX or  ITEM_VALUE< TABLE_VALUE
% 1: ITEM_INDEX = TABLE_INDEX and ITEM_VALUE>=TABLE_VALUE
element_greatereq(ITEM, TABLE) :-
        ITEM = [[index-ITEM_INDEX, value-ITEM_VALUE]],
        element_greatereq_signature(TABLE, SIGNATURE, ITEM_INDEX, ITEM_VALUE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

element_greatereq_signature([], [], _, _).
element_greatereq_signature([[index-TABLE_INDEX,value-TABLE_VALUE]|TABLEs], [S|Ss],
                            ITEM_INDEX, ITEM_VALUE) :-
        ((ITEM_INDEX #= TABLE_INDEX) #/\ (ITEM_VALUE #>= TABLE_VALUE)) #<=> S,
        element_greatereq_signature(TABLEs, Ss, ITEM_INDEX, ITEM_VALUE).
```

A.16 `element_lesseq(ITEM,TABLE)` [25]

`ITEM.value` is less than or equal to one of the entries (i.e. the `value` attribute) of the table `TABLE`.

```
?- element_lesseq([[index-3, value-1]],
                  [[index-1, value-6],[index-2, value-9],
                   [index-3, value-2],[index-4, value-9]]).

% 0: ITEM_INDEX=\=TABLE_INDEX or  ITEM_VALUE> TABLE_VALUE
% 1: ITEM_INDEX = TABLE_INDEX and ITEM_VALUE=<TABLE_VALUE
element_lesseq(ITEM, TABLE) :-
        ITEM = [[index-ITEM_INDEX, value-ITEM_VALUE]],
        element_lesseq_signature(TABLE, SIGNATURE, ITEM_INDEX, ITEM_VALUE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

element_lesseq_signature([], [], _, _).
element_lesseq_signature([[index-TABLE_INDEX,value-TABLE_VALUE]|TABLEs], [S|Ss],
                         ITEM_INDEX, ITEM_VALUE) :-
        ((ITEM_INDEX #= TABLE_INDEX) #/\ (ITEM_VALUE #=< TABLE_VALUE)) #<=> S,
        element_lesseq_signature(TABLEs, Ss, ITEM_INDEX, ITEM_VALUE).
```

A.17 `element_sparse(ITEM,TABLE,DEFAULT)` [17]

`ITEM.value` is equal to one of the entries of the table `TABLE` or to the default value `DEFAULT` if the entry `ITEM.index` does not exist in `TABLE`.

```
?- element_sparse([[index-2, value-5]],
                  [[index-1, value-6],[index-2, value-5],
                   [index-4, value-2],[index-8, value-9]], 5).
true.

% 0: ITEM_INDEX=\=TABLE_INDEX and ITEM_VALUE=\=DEFAULT
% 1: ITEM_INDEX = TABLE_INDEX and ITEM_VALUE = TABLE_VALUE
% 2: ITEM_INDEX=\=TABLE_INDEX and ITEM_VALUE = DEFAULT
element_sparse(ITEM, TABLE, DEFAULT) :-
        ITEM = [[index-ITEM_INDEX, value-ITEM_VALUE]],
        element_sparse_signature(TABLE, SIGNATURE, ITEM_INDEX, ITEM_VALUE, DEFAULT),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(d),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t),
                   arc(s,2,d),
                   arc(d,1,t),
                   arc(d,2,d),
                   arc(d,$,t)],
                  [],[],[]).

element_sparse_signature([], [], _, _, _).
element_sparse_signature([[index-TABLE_INDEX,value-TABLE_VALUE]|TABLEs], [S|Ss],
                         ITEM_INDEX, ITEM_VALUE, DEFAULT) :-
  S in 0..2,
  (ITEM_INDEX #\=TABLE_INDEX #/\ ITEM_VALUE#\=DEFAULT      ) #<=> S#=0,
  (ITEM_INDEX #\=TABLE_INDEX #/\ ITEM_VALUE#= DEFAULT      ) #<=> S#=2,
  (ITEM_INDEX #= TABLE_INDEX #/\ ITEM_VALUE #= TABLE_VALUE) #<=> S#=1,
  element_sparse_signature(TABLEs, Ss, ITEM_INDEX, ITEM_VALUE, DEFAULT).
```

A.18 `global_contiguity(VARIABLES)` [23]

Enforce all variables of the `VARIABLES` collection to be assigned to 0 or 1. In addition, all variables assigned to value 1 appear contiguously.

```
?- global_contiguity([[var-0],[var-1],[var-1],[var-0]]).
true.

% 0: VAR=0
% 1: VAR=1
global_contiguity(VARIABLES) :-
        col_to_list(VARIABLES, LIST_VARIABLES),
        automaton(LIST_VARIABLES, _,
                  LIST_VARIABLES, 0..1,
                  [source(s),node(n),node(z),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,n),
                   arc(s,$,t),
                   arc(n,0,z),
                   arc(n,1,n),
                   arc(n,$,t),
                   arc(z,0,z),
                   arc(z,$,t)],
                  [], [], []).
```

A.19 group(MIN_SIZE,MAX_SIZE,NGROUP,VARIABLES,VALUES) [17]

Let n be the number of variables of the collection VARIABLES. Let $X_i, X_{i+1}, \ldots, X_j$ $(1 \leq i \leq j \leq n)$ be consecutive variables of the collection of variables VARIABLES such that all the following conditions simultaneously apply:

- All variables $X_i, \ldots, X_j$ take their value in the set of values VALUES,
- $i = 1$ or X_{i-1} does not take a value in VALUES,
- $j = n$ or X_{j+1} does not take a value in VALUES.

We call such a set of variables a *group*. The constraint group is true if all the following conditions hold:

- There are exactly NGROUP groups of variables,
- MIN_SIZE is the number of variables of the smallest group,
- MAX_SIZE is the number of variables of the largest group.

```
?- group(1, 2, 2,
         [[var-2],[var-8],[var-1],[var-7],[var-4],[var-5],[var-1],[var-1],[var-1]],
         [[val-0],[val-2],[val-4],[val-6],[val-8]]).
true.

% 0: not_in(VAR,VALUES)
% 1: in(VAR,VALUES)
group(MIN_SIZE, MAX_SIZE, NGROUP, VARIABLES, VALUES) :-
        group_max_size(MAX_SIZE, VARIABLES, VALUES),
        group_min_size(MIN_SIZE, VARIABLES, VALUES),
        group_ngroup(NGROUP, VARIABLES, VALUES).

group_max_size(MAX_SIZE, VARIABLES, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s,[C       ,D+1]),
                   arc(s,0,s,[max(C,D),0  ]),
                   arc(s,$,t,[max(C,D),D  ])],
                  [C,D],[0,0],[MAX_SIZE,_]).

group_min_size(MIN_SIZE, VARIABLES, VALUES) :-
        length(VARIABLES, NVAR),
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(j),node(k),sink(t)],
                  [arc(s,0,s              ),
                   arc(s,1,j,[NVAR    ,D  ]),
                   arc(s,$,t              ),
                   arc(j,1,j,[C       ,D+1]),
                   arc(j,0,k,[min(C,D),D  ]),
                   arc(j,$,t,[min(C,D),D  ]),
                   arc(k,0,k              ),
                   arc(k,1,j,[C       ,1  ]),
                   arc(k,$,t,[min(C,D),D  ])],
                  [C,D],[0,1],[MIN_SIZE,_]).

group_ngroup(NGROUP, VARIABLES, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(i),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,i,[C+1]),
                   arc(s,$,t      ),
                   arc(i,1,i      ),
                   arc(i,0,s      ),
                   arc(i,$,t      )],
                  [C],[0],[NGROUP]).

group_signature([], [], _).
group_signature([[var-VAR]|VARs], [S|Ss], SET_OF_VALUES) :-
        VAR in_set SET_OF_VALUES #<=> S,
        group_signature(VARs, Ss, SET_OF_VALUES).
```

A.20 `group_skip_isolated_item(MIN_SIZE,MAX_SIZE,NGROUP,VARIABLES,VALUES)`

Let n be the number of variables of the collection VARIABLES. Let $X_i, X_{i+1}, \ldots, X_j$ $(1 \leq i < j \leq n)$ be consecutive variables of the collection of variables VARIABLES such that the following conditions apply:

- All variables $X_i, \ldots, X_j$ take their value in the set of values VALUES,
- $i = 1$ or X_{i-1} does not take a value in VALUES,
- $j = n$ or X_{j+1} does not take a value in VALUES.

We call such a set of variables a *group*. The constraint `group_skip_isolated_item` is true if all the following conditions hold:

- There are exactly NGROUP groups of variables,
- The number of variables of the smallest group is MIN_SIZE,
- The number of variables of the largest group is MAX_SIZE.

Inspired by `group`.

```
?- group_skip_isolated_item(2, 2, 1,
                            [[var-2],[var-8],[var-1],[var-7],[var-4],
                             [var-5],[var-1],[var-1],[var-1]],
                            [[val-0],[val-2],[val-4],[val-6],[val-8]]).
true.

% 0: not_in(VAR,VALUES)
% 1: in(VAR,VALUES)
group_skip_isolated_item(MIN_SIZE, MAX_SIZE, NGROUP, VARIABLES, VALUES) :-
        group_skip_isolated_item_max_size(MAX_SIZE, VARIABLES, VALUES),
        group_skip_isolated_item_min_size(MIN_SIZE, VARIABLES, VALUES),
        group_skip_isolated_item_ngroup(NGROUP, VARIABLES, VALUES).

group_skip_isolated_item_max_size(MAX_SIZE, VARIABLES, VALUES) :-
```

```
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_skip_isolated_item_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(i),sink(t)],
                  [arc(s,0,s               ),
                   arc(s,1,i,[C,1]         ),
                   arc(s,$,t               ),
                   arc(i,0,s,[max(C,D),D  ]),
                   arc(i,1,i,[C       ,D+1]),
                   arc(i,$,t,[max(C,D),D  ])],
                  [C,D],[0,0],[MAX_SIZE,_]).

group_skip_isolated_item_min_size(MIN_SIZE, VARIABLES, VALUES) :-
        length(VARIABLES, NVAR),
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_skip_isolated_item_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(j),node(k),node(l),node(m),sink(t)],
                  [arc(s,0,s               ),
                   arc(s,1,j               ),
                   arc(s,$,t               ),
                   arc(j,0,s               ),
                   arc(j,1,k,[NVAR    ,D  ]),
                   arc(j,$,t               ),
                   arc(k,1,k,[C       ,D+1]),
                   arc(k,0,l,[min(C,D),D  ]),
                   arc(k,$,t,[min(C,D),D  ]),
                   arc(l,0,l               ),
                   arc(l,1,m               ),
                   arc(l,$,t,[min(C,D),D  ]),
                   arc(m,0,l               ),
                   arc(m,1,k,[C       ,2  ]),
                   arc(m,$,t,[min(C,D),D  ])],
                  [C,D],[0,2],[MIN_SIZE,_]).

group_skip_isolated_item_ngroup(NGROUP, VARIABLES, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        group_skip_isolated_item_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(i),node(j),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,i      ),
                   arc(s,$,t      ),
                   arc(i,0,s      ),
                   arc(i,1,j,[C+1]),
                   arc(i,$,t      ),
                   arc(j,1,j      ),
                   arc(j,0,s      ),
                   arc(j,$,t      )],
                  [C],[0],[NGROUP]).

group_skip_isolated_item_signature([], [], _).
group_skip_isolated_item_signature([[var-VAR]|VARs], [S|Ss], SET_OF_VALUES) :-
        VAR in_set SET_OF_VALUES #<=> S,
        group_skip_isolated_item_signature(VARs, Ss, SET_OF_VALUES).
```

A.21 in_(VAR,VALUES)

Enforce the domain variable VAR to take a value within the values described by the VALUES collection.

```
?- in_(3, [[val-1], [val-3]]).
true.

% 0: VAR=\=VAL
% 1: VAR=VAL
in_(VAR, VALUES) :-
        in_signature(VALUES, SIGNATURE, VAR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

in_signature([], [], _).
in_signature([[val-VAL]|VALs], [S|Ss], VAR) :-
        VAR #= VAL #<=> S,
        in_signature(VALs, Ss, VAR).
```

A.22 in_same_partition(VAR1,VAR2,PARTITIONS)

Enforce VAR1 and VAR2 to be respectively assigned to values v_1 and v_2 that both belong to a same partition of the collection PARTITIONS.

```
?- in_same_partition(6,2,[[p-[[val-1], [val-3]]],
                          [p-[[val-4]         ]],
                          [p-[[val-2], [val-6]]]]).
true.

% 0: not_in(VAR1,VALUES) or not_in(VAR2,VALUES)
% 1:     in(VAR1,VALUES) and    in(VAR2,VALUES)
in_same_partition(VAR1, VAR2, PARTITIONS) :-
        in_same_partition_signature(PARTITIONS, SIGNATURE, VAR1, VAR2),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,t)],
                  [],[],[]).

in_same_partition_signature([], [], _, _).
in_same_partition_signature([[p-VALUES]|PARTITIONs], [S|Ss], VAR1, VAR2) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        ((VAR1 in_set SET_OF_VALUES) #/\ (VAR2 in_set SET_OF_VALUES)) #<=> S,
        in_same_partition_signature(PARTITIONs, Ss, VAR1, VAR2).
```

A.23 inflexion(N,VARIABLES)

N is equal to the number of times that the following conjunctions of constraints hold:

- X_i CTR $X_{i+1} \wedge X_i \neq X_{i+1}$,
- $X_{i+1} = X_{i+2} \wedge \cdots \wedge X_{j-2} = X_{j-1}$,
- $X_{j-1} \neq X_j \wedge X_{j-1} \neg$ CTR X_j.

where X_k is the k^{th} item of the VARIABLES collection and $1 \leq i, i+2 \leq j, j \leq n$.

```
?- inflexion(4,[[var-3],[var-3],[var-1],[var-4],[var-5],[var-5],
                [var-6],[var-5],[var-5],[var-6],[var-3]]).
true.

% 0: VAR1>VAR2
% 1: VAR1=VAR2
% 2: VAR1<VAR2
inflexion(N, VARIABLES) :-
        inflexion_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(i),node(j),sink(t)],
                  [arc(s,1,s      ),
                   arc(s,2,i      ),
                   arc(s,0,j      ),
                   arc(s,$,t      ),
                   arc(i,1,i      ),
                   arc(i,2,i      ),
                   arc(i,0,j,[C+1]),
                   arc(i,$,t      ),
                   arc(j,1,j      ),
                   arc(j,0,j      ),
                   arc(j,2,i,[C+1]),
                   arc(j,$,t      )],
                  [C],[0],[N]).

inflexion_signature([], []).
inflexion_signature([_], []).
inflexion_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss]) :-
        S in 0..2,
        VAR1 #> VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #< VAR2 #<=> S #= 2,
        inflexion_signature([[var-VAR2]|VARs], Ss).
```

A.24 **lex_different(VECTOR1,VECTOR2)**

Vectors VECTOR1 and VECTOR2 differ from at least one component.

```
?- lex_different([[var-5],[var-2],[var-7],[var-1]],
                 [[var-5],[var-3],[var-7],[var-1]]).
true.

% 0: VAR1=\=VAR2
% 1: VAR1=VAR2
lex_different(VECTOR1, VECTOR2) :-
        lex_different_signature(VECTOR1, VECTOR2, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s),
                   arc(s,0,t)],
                  [], [], []).

lex_different_signature([], [], []).
lex_different_signature([[var-VAR1]|Xs], [[var-VAR2]|Ys], [S|Ss]) :-
        VAR1 #= VAR2 #<=> S,
        lex_different_signature(Xs, Ys, Ss).
```

A.25 **lex_lesseq(VECTOR1,VECTOR2)** [18]

VECTOR1 is lexicographically less than or equal to VECTOR2.
Given two vectors, $\overrightarrow{X}$ and $\overrightarrow{Y}$ of n components, $\langle X_0,\ldots,X_n\rangle$ and $\langle Y_0,\ldots,Y_n\rangle$, $\overrightarrow{X}$ is *lexicographically less than or equal to* $\overrightarrow{Y}$ iff $n = 0$ or $X_0 < Y_0$ or $X_0 = Y_0$ and $\langle X_1,\ldots,X_n\rangle$ is lexicographically less than or equal to $\langle Y_1,\ldots,Y_n\rangle$.

```
?- lex_lesseq([[var-5], [var-2], [var-3], [var-1]],
              [[var-5], [var-2], [var-6], [var-2]]).
```

```
true.

% 1: VAR1 < VAR2
% 2: VAR1 = VAR2
% 3: VAR1 > VAR2
lex_lesseq(VECTOR1, VECTOR2) :-
        lex_lesseq_signature(VECTOR1, VECTOR2, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 1..3,
                  [source(s),sink(t)],
                  [arc(s,1,t),
                   arc(s,2,s),
                   arc(s,$,t)],
                  [], [], []).

lex_lesseq_signature([], [], []).
lex_lesseq_signature([[var-VAR1]|Xs], [[var-VAR2]|Ys], [S|Ss]) :-
        S in 1..3,
        VAR1 #< VAR2 #<=> S #= 1,
        VAR1 #= VAR2 #<=> S #= 2,
        VAR1 #> VAR2 #<=> S #= 3,
        lex_lesseq_signature(Xs, Ys, Ss).
```

A.26 longest_change(SIZE,VARIABLES,CTR)

SIZE is the maximum number of consecutive variables of the collection VARIABLES for which constraint CTR holds in an uninterrupted way. We count a change when X CTR Y holds; X and Y are two consecutive variables of the collection VARIABLES. Derived from change.

```
?- longest_change(4, [[var-8],[var-8],[var-3],[var-4],[var-1],
                              [var-1],[var-5],[var-5],[var-2]], =\=).
true.

% CTR: =
% 0: VAR1=\=VAR2
% 1: VAR1=VAR2$\MMI$ is the index of the variables corresponding to the
maximum value of the collection of variables $\MVARIABLES$.
%
% CTR: =\=
% 0: VAR1=VAR2
% 1: VAR1=\=VAR2
%
% CTR: <
% 0: VAR1>=VAR2
% 1: VAR1<VAR2
%
% CTR: >=
% 0: VAR1<VAR2
% 1: VAR1>=VAR2
%
% CTR: >
% 0: VAR1=<VAR2
% 1: VAR1>VAR2
%
% CTR: =<
% 0: VAR1>VAR2
% 1: VAR1=<VAR2
longest_change(SIZE, VARIABLES, CTR) :-
        longest_change_signature(VARIABLES, SIGNATURE, CTR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s,[C       ,D+1]),
                   arc(s,0,s,[max(C,D),1  ]),
                   arc(s,$,t,[max(C,D),D  ])],
                  [C,D],[0,1],[SIZE,_]).
```

```
longest_change_signature([], [], _).
longest_change_signature([_], [], _) :- !.
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =) :- !,
        VAR1 #= VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, =).
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =\=) :- !,
        VAR1 #\= VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, =\=).
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], <) :- !,
        VAR1 #< VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, <).
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >=) :- !,
        VAR1 #>= VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, >=).
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], >) :- !,
        VAR1 #> VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, >).
longest_change_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], =<) :- !,
        VAR1 #=< VAR2 #<=> S,
        longest_change_signature([[var-VAR2]|VARs], Ss, =<).
```

A.27 max_index(MAX_INDEX,VARIABLES)

MAX_INDEX is the index of the variables corresponding to the maximum value of the collection of variables VARIABLES.

```
?- max_index(3, [[index-1, var-3],[index-2, var-2],[index-3, var-7],
                                  [index-4, var-2],[index-5, var-6]]).
true.

% 0: VAR
max_index(MAX_INDEX, VARIABLES) :-
        length(VARIABLES, N),
        length(SIGNATURE, N),
        domain(SIGNATURE, 0, 0),
        max_index_signature(VARIABLES, 1, VARS, SIGNATURE),
        automaton(VARS, VAR, SIGNATURE, 0..0,
                  [source(s),sink(t)],
                  [arc(s,0,s,(VAR#=<M -> [M,I,J+1] ; VAR#>M -> [VAR,J+1,J+1])),
                   arc(s,$,t)                                              ],
                  [M,I,J],[-1000000,0,0],[_,MAX_INDEX,_]).

max_index_signature([], _, [], []).
max_index_signature([[index-I, var-V]|VARs], I, [V|Vs], [0|Ss]) :-
        J is I+1,
        max_index_signature(VARs, J, Vs, Ss).
```

A.28 maximum(MAX,VARIABLES)

MAX is the maximum value of the collection of domain variables VARIABLES.

```
?- maximum(7, [[var-3],[var-2],[var-7],[var-2],[var-6]]).
true.

% 0: MAX>VAR
% 1: MAX=VAR
% 2: MAX<VAR
maximum(MAX, VARIABLES) :-
        maximum_signature(VARIABLES, SIGNATURE, MAX),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(e),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,e),
                   arc(e,1,e),
                   arc(e,0,e),
                   arc(e,$,t)],
                  [],[],[]).

maximum_signature([], [], _).
maximum_signature([[var-VAR]|VARs], [S|Ss], MAX) :-
        S in 0..2,
        MAX #> VAR #<=> S #= 0,
        MAX #= VAR #<=> S #= 1,
        MAX #< VAR #<=> S #= 2,
        maximum_signature(VARs, Ss, MAX).
```

A.29 not_all_equal(VARIABLES)

The variables of the collection VARIABLES should take more than one single value.

```
?- not_all_equal([[var-3],[var-1],[var-3],[var-3],[var-3]]).
true.

% 0: VAR1=\=VAR2
% 1: VAR1=VAR2
not_all_equal(VARIABLES) :-
        length(VARIABLES,N),
        N > 1,
        not_all_equal_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s),
                   arc(s,0,t)],
                  [],[],[]).

not_all_equal_signature([], []).
not_all_equal_signature([_], []).
not_all_equal_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss]) :-
        VAR1 #= VAR2 #<=> S,
        not_all_equal_signature([[var-VAR2]|VARs], Ss).
```

A.30 not_in(VAR,VALUES)

Remove the values of the VALUES collection from domain variable VAR.

```
?- not_in(2, [[val-1],[val-3]]).
true.

% 0: VAR=\=VAL
% 1: VAR=VAL
not_in(VAR, VALUES) :-
        not_in_signature(VALUES, SIGNATURE, VAR),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,$,t)],
                  [],[],[]).

not_in_signature([], [], _).
not_in_signature([[val-VAL]|VALs], [S|Ss], VAR) :-
        VAR #= VAL #<=> S,
        not_in_signature(VALs, Ss, VAR).
```

A.31 `pattern(VARIABLES)` [11]

We quote the definition from the original paper [11, page 157] introducing the `pattern` constraint.
We call a k-pattern any sequence of k elements such that no two successive elements have the same value. Consider a set $V = \{v_1, v_2, \ldots, v_m\}$ and a sequence $\mathbf{s} = \langle s_1, s_2, \ldots, s_n\rangle$ of elements of V. Consider now the sequence $\langle v_{i1}, v_{i2}, \ldots, v_{il}\rangle$ of the types of the successive stretches that appear in $\mathbf{s}$. Let $\mathcal{P}$ be a set of k-pattern. Vector $\mathbf{s}$ satisfies $\mathcal{P}$ if and only if every subsequence of k elements in $\langle v_{i1}, v_{i2}, \ldots, v_{il}\rangle$ belongs to $\mathcal{P}$.

```
?- pattern([[var-0],[var-2],[var-0],[var-3]]).
true.

% 0: VAR=0 (o)
% 1: VAR=1 (d)
% 2: VAR=2 (e)
% 3: VAR=3 (n)
% PESANT EXAMPLE OF CP2003
pattern(VARIABLES) :-
        pattern_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..3,
                  [source(s), node(o)  , node(d)  , node(e)  , node(n)  ,
                   node(od) , node(oe) , node(on) , node(do) , node(eo) , node(no) ,
                   node(odo), node(oeo), node(ono), node(dod), node(doe), node(eoe),
                   node(eon), node(nod), node(non), sink(t)                        ],
                  [arc(s  ,0,o  ),
                   arc(s  ,1,d  ),
                   arc(s  ,2,e  ),
                   arc(s  ,3,n  ),
                   arc(o  ,0,o  ),
                   arc(o  ,1,od ),
                   arc(o  ,2,oe ),
                   arc(o  ,3,on ),
                   arc(d  ,1,d  ),
                   arc(d  ,0,do ),
                   arc(e  ,2,e  ),
                   arc(e  ,0,eo ),
                   arc(n  ,3,n  ),
                   arc(n  ,0,no ),
                   arc(od ,1,od ),
                   arc(od ,0,odo),
                   arc(oe ,2,oe ),
                   arc(oe ,0,oeo),
                   arc(on ,3,on ),
                   arc(on ,0,ono),
                   arc(do ,0,do ),
                   arc(do ,1,dod),
                   arc(do ,2,doe),
                   arc(eo ,0,eo ),
                   arc(eo ,2,eoe),
                   arc(eo ,3,eon),
                   arc(no ,0,no ),
                   arc(no ,1,nod),
                   arc(no ,3,non),
                   arc(odo,0,odo),
                   arc(odo,1,dod),
                   arc(odo,2,doe),
                   arc(odo,$,t  ),
                   arc(oeo,0,oeo),
                   arc(oeo,2,eoe),
                   arc(oeo,3,eon),
                   arc(oeo,$,t  ),
                   arc(ono,0,ono),
                   arc(ono,1,nod),
                   arc(ono,3,non),
                   arc(ono,$,t  ),
                   arc(dod,1,dod),
                   arc(dod,0,odo),
                   arc(dod,$,t  ),
                   arc(doe,2,doe),
                   arc(doe,0,oeo),
                   arc(doe,$,t  ),
                   arc(eoe,2,eoe),
                   arc(eoe,0,oeo),
                   arc(eoe,$,t  ),
                   arc(eon,3,eon),
                   arc(eon,0,ono),
                   arc(eon,$,t  ),
                   arc(nod,1,nod),
                   arc(nod,0,odo),
                   arc(nod,$,t  ),
                   arc(non,3,non),
                   arc(non,0,ono),
                   arc(non,$,t  )],
                  [],[],[]).

pattern_signature([], []).
pattern_signature([[var-VAR]|VARs], [VAR|Ss]) :-
        pattern_signature(VARs, Ss).
```

A.32 peak(N,VARIABLES)

A variable V_k $(1 < k < m)$ of the sequence of variables VARIABLES $= V_1, \ldots, V_m$ is a *peak* if and only if there exist an i $(1 < i \leq k)$ such that $V_{i-1} < V_i$ and $V_i = V_{i+1} = \ldots = V_k$ and $V_k > V_{k+1}$. N is the total number of peaks of the sequence of variables VARIABLES.

```
?- peak(2, [[var-3],[var-3],[var-1],[var-4],[var-5],[var-5],
            [var-8],[var-5],[var-5],[var-6],[var-6],[var-3]]).
true.

% 0: VAR1>VAR2
% 1: VAR1=VAR2
% 2: VAR1<VAR2
peak(N, VARIABLES) :-
        peak_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(u),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s      ),
                   arc(s,2,u      ),
                   arc(s,$,t      ),
                   arc(u,0,s,[C+1]),
                   arc(u,1,u      ),
                   arc(u,2,u      ),
                   arc(u,$,t      )],
                  [C],[0],[N]).

peak_signature([], []).
peak_signature([_], []).
peak_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss]) :-
        S in 0..2,
        VAR1 #> VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #< VAR2 #<=> S #= 2,
        peak_signature([[var-VAR2]|VARs], Ss).
```

A.33 sequence_folding(LETTERS)

Express the fact that a sequence is folded in a way that no crossing occurs. A sequence is modelled by a collection of letters. For each letter l_1 of a sequence, we indicate the next letter l_2 located after l_1 which is directly in contact with l_1 (l_1 itself if such a letter does not exist). Derived by Justin Pearson and motivated by RNA folding [15].

```
?- sequence_folding([[index-1, next-1],[index-2, next-8],[index-3, next-3],
                     [index-4, next-5],[index-5, next-5],[index-6, next-7],
                     [index-7, next-7],[index-8, next-8],[index-9, next-9]]).
true.

% 0: INDEX1<INDEX2 and INDEX1=<NEXT1 and INDEX2=<NEXT2 and NEXT1=<INDEX2
% 1: INDEX1<INDEX2 and INDEX1=<NEXT1 and INDEX2=<NEXT2 and NEXT1 >INDEX2 and NEXT2=<NEXT1
sequence_folding(LETTERS) :-
        sequence_folding_signature(LETTERS, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,s),
                   arc(s,$,t)],
                  [],[],[]).

sequence_folding_signature([], []).
sequence_folding_signature([_], []).
sequence_folding_signature([L1,L2|R], S) :-
        sequence_folding_signature([L2|R], L1, S1),
        sequence_folding_signature([L2|R], S2),
        append(S1, S2, S).

sequence_folding_signature([], _, []).
sequence_folding_signature([L2|R], L1, [S|Ss]) :-
        L1 = [index-INDEX1,next-NEXT1],
        L2 = [index-INDEX2,next-NEXT2],
        INDEX1#<INDEX2 #/\ INDEX1#=<NEXT1 #/\
        INDEX2#=<NEXT2 #/\ NEXT1#=<INDEX2 #<=> S #= 0,
        INDEX1#<INDEX2 #/\ INDEX1#=<NEXT1 #/\
        INDEX2#=<NEXT2 #/\ NEXT1#>INDEX2  #/\ NEXT2#=<NEXT1 #<=> S #= 1,
        sequence_folding_signature(R, L1, Ss).
```

A.34 sliding_card_skip0(ATLEAST,ATMOST,VARIABLES,VALUES)

Let n be the total number of variables of the collection VARIABLES. A *maximum non-zero set of consecutive variables* $X_i..X_j (1 \leq i \leq j \leq n)$ is defined in the following way:

- All variables $X_i, \ldots, X_j$ take a non-zero value,
- $i = 1$ or X_{i-1} is equal to 0,
- $j = n$ or X_{j+1} is equal to 0.

Enforces that each maximum non-zero set of consecutive variables of the collection VARIABLES contains at least ATLEAST and at most ATMOST values from the collection of values VALUES.

```
?- sliding_card_skip0(2, 3,
                      [[var-0],[var-7],[var-2],[var-9],[var-0],
                       [var-0],[var-9],[var-4],[var-9]],
                      [[val-7],[val-9]]).
true.

% 0: VAR=0
% 1: VAR=\=0 and not_in(VAR,VALUES)
% 2: VAR=\=0 and in(VAR,VALUES)
sliding_card_skip0(ATLEAST, ATMOST, VARIABLES, VALUES) :-
        col_to_list(VALUES, LIST_VALUES),
        list_to_fdset(LIST_VALUES, SET_OF_VALUES),
        sliding_card_skip0_signature(VARIABLES, SIGNATURE, SET_OF_VALUES),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(i),sink(t)],
                  [arc(s,0,s                              ),
                   arc(s,1,i,[0]                          ),
                   arc(s,2,i,[1]                          ),
                   arc(s,$,t                              ),
                   arc(i,0,s,(C in ATLEAST..ATMOST -> [C])),
                   arc(i,1,i                             ),
                   arc(i,2,i,[C+1]                       ),
                   arc(i,$,t,(C in ATLEAST..ATMOST -> [C]))],
                  [C],[0],[_]).

sliding_card_skip0_signature([], [], _).
sliding_card_skip0_signature([[var-VAR]|VARs], [S|Ss], SET_OF_VALUES) :-
        VAR #\= 0               #<=> NZ,
        VAR in_set SET_OF_VALUES #<=> In,
        S in 0..2,
        S #= max(2*NZ + In - 1, 0),
        sliding_card_skip0_signature(VARs, Ss, SET_OF_VALUES).
```

A.35 `smooth(NCHANGE,TOLERANCE,VARIABLES)`

NCHANGE is the number of times that $|X - Y| >$ TOLERANCE holds; X and Y correspond to consecutive variables of the collection VARIABLES.

```
?- smooth(1, 2, [[var-1],[var-3],[var-4],[var-5],[var-2]]).
true.

% 0: |VAR1-VAR2|=<TOLERANCE
% 1: |VAR1-VAR2|>TOLERANCE
smooth(NCHANGE, TOLERANCE, VARIABLES) :-
        smooth_signature(VARIABLES, SIGNATURE, TOLERANCE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s,[C+1]),
                   arc(s,0,s      ),
                   arc(s,$,t      )],
                  [C],[0],[NCHANGE]).

smooth_signature([], [], _).
smooth_signature([_], [], _).
smooth_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss], TOLERANCE) :-
        abs(VAR1-VAR2) #> TOLERANCE #<=> S #= 1,
        smooth_signature([[var-VAR2]|VARs], Ss, TOLERANCE).
```

A.36 `top(N,VARIABLES)` [8]

A variable V_k $(1 \leq k \leq m)$ of the sequence of variables VARIABLES $= V_1, \ldots, V_m$ is a *top* if and only if there exist an i $(1 \leq i \leq k)$ such that $V_{i-1} < V_i$ and $V_i = V_{i+1} = \ldots = V_k$ and $V_k > V_{k+1}$ with convention that $V_0 = V_{m+1} = 0$. N is the total number of tops of the sequence of variables VARIABLES.

```
?- top(3, [[var-3],[var-3],[var-1],[var-4],[var-5],[var-5],
           [var-6],[var-5],[var-5],[var-6],[var-3]]).
true.

% 0: VAR1>VAR2
% 1: VAR1=VAR2
% 2: VAR1<VAR2
top(N, VARIABLES) :-
        top_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(u),sink(t)],
                  [arc(s,0,u,[C+1]),
                   arc(s,1,s      ),
                   arc(s,2,s      ),
                   arc(s,$,t,[C+1]),
                   arc(u,1,u      ),
                  arc(u,0,u      ),
                  arc(u,2,s      ),
                  arc(u,$,t      )],
                 [C],[0],[N]).

top_signature([], []).
top_signature([_], []).
top_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss]) :-
        S in 0..2,
        VAR1 #> VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #< VAR2 #<=> S #= 2,
        top_signature([[var-VAR2]|VARs], Ss).
```

A.37 **`two_quad_are_in_contact(QUADRANGLE1,QUADRANGLE2)`** [17]

Enforce the following conditions on two quadrangles QUADRANGLE1 and QUADRANGLE2:

- For all dimensions i, except one dimension, the projections of QUADRANGLE1 and QUADRANGLE2 on i have a non-empty intersection.
- For all dimensions i, the distance between the projections of QUADRANGLE1 and QUADRANGLE2 on i is equal to 0.

```
?- two_quad_are_in_contact([[ori-1,siz-3,end-4], [ori-5,siz-2,end-7]],
                           [[ori-3,siz-2,end-5], [ori-2,siz-3,end-5]]).
true.

% 0: SIZ1>0 and SIZ2>0 and  END1>ORI2 and END2>ORI1
% 1: SIZ1>0 and SIZ2>0 and (END1=ORI2 or  END2=ORI1)
two_quad_are_in_contact(QUADRANGLE1, QUADRANGLE2) :-
        two_quad_are_in_contact_signature(QUADRANGLE1, QUADRANGLE2, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),node(z),sink(t)],
                  [arc(s,0,s),
                   arc(s,1,z),
                   arc(z,0,z),
                   arc(z,$,t)],
                  [],[],[]).

two_quad_are_in_contact_signature([], [], []).
two_quad_are_in_contact_signature([[ori-ORI1,siz-SIZ1,end-END1]|Q1],
                                  [[ori-ORI2,siz-SIZ2,end-END2]|Q2], [S|Ss]) :-
        S in 0..1,
        (SIZ1#>0 #/\ SIZ2#>0 #/\  END1#>ORI2 #/\ END2#>ORI1 ) #<=> S#=0,
        (SIZ1#>0 #/\ SIZ2#>0 #/\ (END1#=ORI2 #\/ END2#=ORI1)) #<=> S#=1,
        two_quad_are_in_contact_signature(Q1, Q2, Ss).
```

A.38 **`two_quad_do_not_overlap(QUADRANGLE1,QUADRANGLE2)`** [17]

For two quadrangles QUADRANGLE1 and QUADRANGLE2 enforce that there exist at least one dimension i such that the projections on i of QUADRANGLE1 and QUADRANGLE2 do not overlap.

```
?- two_quad_do_not_overlap([[ori-2,siz-2,end-4], [ori-1,siz-3,end-4]],
                           [[ori-4,siz-4,end-8], [ori-3,siz-3,end-6]]).
true.

% 0: SIZ1=0 or  SIZ2=0 or  END1=<ORI2 or  END2=<ORI1
% 1: SIZ1>0 and SIZ2>0 and END1 >ORI2 and END2 >ORI1
two_quad_do_not_overlap(QUADRANGLE1, QUADRANGLE2) :-
        two_quad_do_not_overlap_signature(QUADRANGLE1, QUADRANGLE2, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..1,
                  [source(s),sink(t)],
                  [arc(s,1,s),
                   arc(s,0,t)],
                  [],[],[]).

two_quad_do_not_overlap_signature([], [], []).
two_quad_do_not_overlap_signature([[ori-ORI1,siz-SIZ1,end-END1]|Q1],
                                  [[ori-ORI2,siz-SIZ2,end-END2]|Q2], [S|Ss]) :-
        ((SIZ1#>0) #/\ (SIZ2#>0) #/\ (END1#>ORI2) #/\ (END2#>ORI1)) #<=> S,
        two_quad_do_not_overlap_signature(Q1, Q2, Ss).
```

A.39 `valley(N,VARIABLES)`

A variable V_k $(1 < k < m)$ of the sequence of variables `VARIABLES` $= V_1, \ldots, V_m$ is a *valley* if and only if there exist an i $(1 < i \leq k)$ such that $V_{i-1} > V_i$ and $V_i = V_{i+1} = \ldots = V_k$ and $V_k < V_{k+1}$. `N` is the total number of valleys of the sequence of variables `VARIABLES`.

```
?- valley(2, [[var-3],[var-3],[var-1],[var-4],[var-5],[var-5],
              [var-6],[var-5],[var-5],[var-6],[var-3]]).
true.

% 0: VAR1<VAR2
% 1: VAR1=VAR2
% 2: VAR1>VAR2
valley(N, VARIABLES) :-
        valley_signature(VARIABLES, SIGNATURE),
        automaton(SIGNATURE, _,
                  SIGNATURE, 0..2,
                  [source(s),node(u),sink(t)],
                  [arc(s,0,s      ),
                   arc(s,1,s      ),
                   arc(s,2,u      ),
                   arc(s,$,t      ),
                   arc(u,0,s,[C+1]),
                   arc(u,1,u      ),
                   arc(u,2,u      ),
                   arc(u,$,t      )],
                  [C],[0],[N]).

valley_signature([], []).
valley_signature([_], []).
valley_signature([[var-VAR1],[var-VAR2]|VARs], [S|Ss]) :-
        S in 0..2,
        VAR1 #< VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #> VAR2 #<=> S #= 2,
        valley_signature([[var-VAR2]|VARs], Ss).
```