Constraints, 10, 339–362, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Reformulation of Global Constraints Based on Constraints Checkers

NICOLAS BELDICEANU Nicolas.Beldiceanu@emn.fr
LINA FRE CNRS 2729, École des Mines de Nantes, FR-44307 Nantes Cedex 3, France

MATS CARLSSON Mats.Carlsson@sics.se
SICS, P.O. Box 1263, SE-164 29 Kista, Sweden

ROMUALD DEBRUYNE Romuald.Debruyne@emn.fr
LINA FRE CNRS 2729, École des Mines de Nantes, FR-44307 Nantes Cedex 3, France

THIERRY PETIT Thierry.Petit@emn.fr
LINA FRE CNRS 2729, École des Mines de Nantes, FR-44307 Nantes Cedex 3, France

Abstract. This article deals 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 reducing the automaton to a conjunction of signature and transition constraints we show how to systematically obtain an automaton reformulation. Under some restrictions on the signature and transition constraints, this reformulation maintains 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 an automaton reformulation 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: global constraints, automata, reformulation

1. Introduction

Developing filtering algorithms for global constraints is usually a very work-intensive and error-prone activity. As a first step toward a methodology for semi-automatic development of filtering algorithms for global constraints, Carlsson and Beldiceanu introduced [1] an approach to designing 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. As shown by the following chronological list of related work, automata were already used for handling constraints:

– Vempaty introduced the idea of representing the solution set of a constraint network by a minimized deterministic finite state automaton [2]. He showed how to use this canonical form to answer queries related to constraints, as satisfiability, validity (i.e., the set of allowed tuples of a constraint), or equivalence between two constraints.

This was essentially done by tracing paths in an acyclic graph derived from the automaton.
- Later, Amilhastre [3] generalized this approach to nondeterministic automata and introduced heuristics to reduce their size. The goal was to obtain a compact representation of the set of solutions of a CSP. That work was applied to configuration problems in [4].
- Boigelot and Wolper [5] also used automata for arithmetic constraints.
- Within the context of global constraints on a finite sequence of variables, the recent work of Pesant [6] also uses a finite automaton for representing the solution set. In this context, it provides a filtering algorithm which maintains arc-consistency.

This article focuses 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 reduce the automaton associated with a global constraint to a conjunction of signature and transition constraints. We characterize some restrictions on the signature and transition constraints under which the filtering induced by this reduction maintains arc-consistency and apply this new methodology to the following problems:

- The design of automaton reformulations for a fairly large set of global constraints.
- The design of automaton reformulations for handling the conjunction of several global constraints.
- The design of constraints between two sequences of variables.

While all previous related work (see Vempaty [2], Amilhastre et al. [4] and Pesant [6]) relies on simple automata and uses an ad-hoc filtering algorithm, our approach is based on automata with counters and reformulation into constraints for which filtering algorithms already exist. Note also that all previous work restricts a transition of the automaton to checking whether or not a given value belongs to the domain of a variable. In contrast, our approach permits to associate any constraint to a transition. As a consequence, we can model concisely a larger class of global constraints and prove properties on the consistency by reasoning directly on the constraint hypergraph. As an illustrative example, consider the lexicographical ordering constraint between two vectors. As shown by Figure 1, we come up with an automaton with two states where a transition constraint corresponds to comparing two domain variables. Now, if we forbid the use of comparison, this would lead to an automaton whose size depends of the number of values in the domains of the variables.

Our second contribution is to provide for a restricted class of automata an automaton reformulation for the relaxed case. This technique relies on the variable based violation cost introduced in [7, 8]. 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 [7]. We come up with an algorithm for computing a

```
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.
```

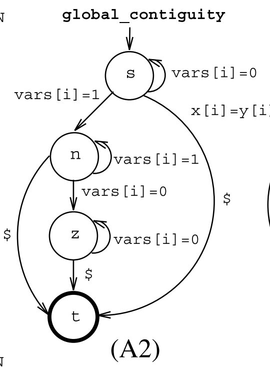
(B1)

```
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.
```

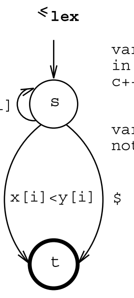
(C1)

```
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.
```

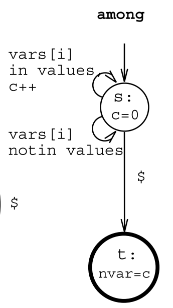
(D1)



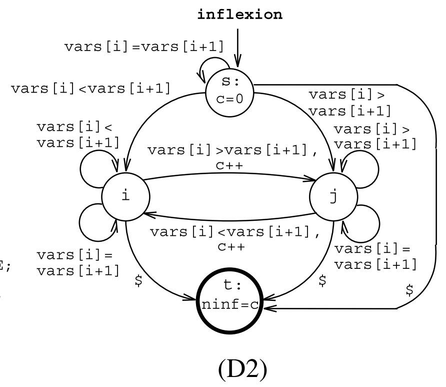
(A2)



(B2)



(C2)



(D2)

Figure 1. Four checkers and their corresponding automata.

sharp bound of the minimum violation cost and with an automaton reformulation for pruning in order to avoid to exceed a given maximum violation cost.

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 an automaton reformulation which exploits the previously introduced automaton. Section 4 describes typical applications of this technique. Finally, for a restricted class of automata, Section 5 provides a filtering algorithm for the relaxed case.

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 concisely expressing 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.[1] Observe that a constraint, like for instance the cycle(n, $[x_1,x_2,\dots,x_m]$) constraint, which enforces that a permutation $[x_1,x_2,\dots,x_m]$ has n cycles, does not belong to this class since it requires to *jump* from one position to another position of the sequence $x_1,x_2,\dots,x_m$. 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.

2.1. 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 [9], minimum [10], pattern [11], global_contiguity [12], lexicographic ordering [13], among [14] and inflexion [15]. 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 $\vec{x} \leq_{\text{lex}} \vec{y}$ over two vectors of variables $\vec{x} = \langle x_0,\dots,x_{n-1}\rangle$ and $\vec{y} = \langle y_0,\dots,y_{n-1}\rangle$ holds iff $n = 0$ or $x_0 < y_0$ or $x_0 = y_0$ and $\langle x_1,\dots,x_{n-1}\rangle \leq_{\text{lex}} \langle y_1,\dots,y_{n-1}\rangle$.
– The among(*nvars*, *vars*, *values*) constraint restricts the number of variables of the sequence of variables *vars* that take their value from a given set *values* to be equal to the variable *nvars*. For instance, among(3, [4, 5, 5, 4, 1], [1, 5, 8]) holds since exactly 3 values of the sequence 45541 are 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 Figure 1 depict the four checkers respectively associated with global_contiguity, $\leq_{\text{lex}}$, among and inflexion.[2] For each checker we observe the following facts:

– Within the checker depicted by part (A1) of Figure 1, the values of the sequence $vars[0],\dots, vars[n-1]$ 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 Figure 1. The automaton takes as input the sequence $vars[0],\dots, vars[n-1]$, 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 Figure 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 Figure 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 Figure 1 counts the number of variables of $vars[0], \ldots, vars[n-1]$ that take their value from *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 Figure 1 describes the automaton corresponding to the code given in part (C1) of the same figure. The automaton uses the counter c, initially set to 0, and takes as input the sequence $vars[0], \ldots, vars[n-1]$. 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 that 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 Figure 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 $vars[0], \ldots, vars[n-1]$. Therefore, the checker depicted in part (D1) of Figure 1 also uses 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 Figure 1. It takes as input the sequence of pairs $\langle vars[0], vars[1]\rangle$, $\langle vars[1], vars[2]\rangle, \ldots, \langle vars[n-2], vars[n-1]\rangle$ 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, if no transition can be triggered, this 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 $\Delta_0, \ldots, \Delta_{m-1}$ denote the sequence of sequences 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 $\Delta_0, \ldots, \Delta_{m-1}$ outside the automaton. To each subset Δ_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 $S_0, \ldots, S_{m-1}$ and $\Delta_0, \ldots, \Delta_{m-1}$ are respectively called the *signature* and the *signature argument* of the constraint. The constraint between S_i and the variables of Δ_i is called the *signature constraint* and is denoted by $\Psi_{\mathcal{C}}(S_i, \Delta_i)$.
- 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 (i.e., counters), assignment statements and control structures are used. This suggested us to consider deterministic finite automata augmented with counters and assignment statements on these counters. 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 counter of the automaton in the final state.

2.2. Defining an Automaton

An automaton $\mathcal{A}$ of a constraint $\mathcal{C}$ is defined by a sextuple

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

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 $\Delta_0, \ldots, \Delta_{m-1}$ of the constraint $\mathcal{C}$. The link between the variables of Δ_i and the variable S_i $(0 \le i < m)$ is done by writing down the signature constraint $\Psi_{\mathcal{C}}(S_i, \Delta_i)$ in such a way that arc-consistency is maintained. In our context this is done by using standard features of the CLP(FD) solver of SICStus Prolog [16] such as arithmetic constraints between two variables, propositional combinators or the global constraints programming interface.

- $\mathcal{C}$*ounters* is the, possibly empty, list of all counters used in the automaton $\mathcal{A}$. Each counter is described by a term t(*Counter*, *Initial Value*, *Final Variable*) where *Counter* is a symbolic name representing the counter, *Initial Value* is an integer giving the value of the counter in the initial state of $\mathcal{A}$, and *Final Variable* 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 source(id), sink(id) or node(id). id is a unique identifier associated with each state. Finally, source (id) and 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 arc(id_1, *label*, id_2) or 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:

- $\mathcal{S}ignature = S_0, S_1, \ldots, S_{n-2}$,
- $\mathcal{S}ignature\mathcal{D}omain = 0..2$,
- $\mathcal{S}ignature\mathcal{A}rg = \langle vars[0], vars[1]\rangle, \ldots, \langle vars[n-2], vars[n-1]\rangle$,
- $\mathcal{C}ounters = \mathrm{t}(c, 0, ninf)$,
- $\mathcal{S}tates = [source(s), node(i), node(j), sink(t)]$,
- $\mathcal{T}ransitions$ = [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)].

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. Automaton Reformulation

The automaton reformulation 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* $\Phi_{\mathcal{C}}\left(Q_i, \overrightarrow{K}_i, S_i, Q_{i+1}, \overrightarrow{K}_{i+1}\right)$ as follows. Q_i is a variable whose values correspond to the states that can be reached at step i. Similarly, $\overrightarrow{K}_i$ is a vector of variables whose values correspond to the potential values of the counters at step i. Assuming that the automaton associated with $\mathcal{C}$ has na arcs $\mathrm{arc}\left(q_1, s_1, q'_1, \overrightarrow{f_1}\left(\overrightarrow{K}\right)\right), \ldots,$ $\mathrm{arc}\left(q_{na}, s_{na}, q'_{na}, \overrightarrow{f_{na}}\left(\overrightarrow{K}\right)\right)$, the transition constraint has the following form:

$$\bigvee_{j=1}^{na}\left[\left(Q_i = q_j\right) \wedge \left(S_i = s_j\right) \wedge \left(Q_{i+1} = q'_j\right) \wedge \left(\overrightarrow{K}_{i+1} = \overrightarrow{f_j}\left(\overrightarrow{K}_i\right)\right)\right]$$

Consider first the `case` when no counter is used, i.e., the constraint is effectively $\Phi_{\mathcal{C}}(Q_i, S_i, Q_{i+1})$. This can be encoded with a ternary relation defined by extension (e.g., SICStus Prolog's `case` [16, page 463], ECLiPSe's Propia [17], or Ilog Solver's table constraint [18]). If that relation maintains arc-consistency, as does `case`, it follows that $\Phi_{\mathcal{C}}$ maintains arc-consistency.

Consider now the case when one counter is used. Then we need to extend the ternary relation by one argument corresponding to j. This argument can be used as an index into a vector $\left[\overrightarrow{f_1}\left(\overrightarrow{K}_i\right), \ldots, \overrightarrow{f_{na}}\left(\overrightarrow{K}_i\right)\right]$, selecting the value that $\overrightarrow{K}_{i+1}$ should equal. Thus to encode $\Phi_{\mathcal{C}}$ we need:

- a 4-ary relation defined by extension,
- *na* arithmetic constraints to compute the vector, and
- an `element` constraint to select a value from the vector.

As an optimization, identical $\overrightarrow{f_j}\left(\overrightarrow{K}_i\right)$ expressions can be merged, yielding a shorter vector and fewer arithmetic constraints. In general, arc-consistency can not be guaranteed for $\Phi_{\mathcal{C}}$.

Finally, consider the case when two or more counters are used. This is a straightforward generalization of the single counter case.

We can then arrive at an automaton reformulation 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 $\Psi_{\mathcal{C}}(S_i, \Delta_i)(0 \leq i < m)$ that relate each signature variables S_i to the variables of its corresponding signature argument Δ_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):

$$\begin{array}{lll}
\Psi_{\mathcal{C}}(S_0, \Delta_0) & \wedge & \Phi_{\mathcal{C}}\left(s, \overrightarrow{K}_0, S_0, Q_1, \overrightarrow{K}_1\right) \wedge \\
\Psi_{\mathcal{C}}(S_1, \Delta_1) & \wedge & \Phi_{\mathcal{C}}\left(Q_1, \overrightarrow{K}_1, S_1, Q_2, \overrightarrow{K}_2\right) \wedge \\
\vdots & & \\
\Psi_{\mathcal{C}}(S_{m-1}, \Delta_{m-1}) & \wedge & \Phi_{\mathcal{C}}\left(Q_{m-1}, \overrightarrow{K}_{m-1}, S_{m-1}, Q_m, \overrightarrow{K}_m\right) \wedge \\
 & & \Phi_{\mathcal{C}}\left(Q_m, \overrightarrow{K}_m, \$, \mathrm{t}, \overrightarrow{K}_{m+1}\right)
\end{array}$$

A couple of examples will help clarify this idea. In these examples, the relation defined by extension is depicted in a compact form as a decision tree. Note that the decision tree needs to correctly handle the case when the terminal state has already been reached.

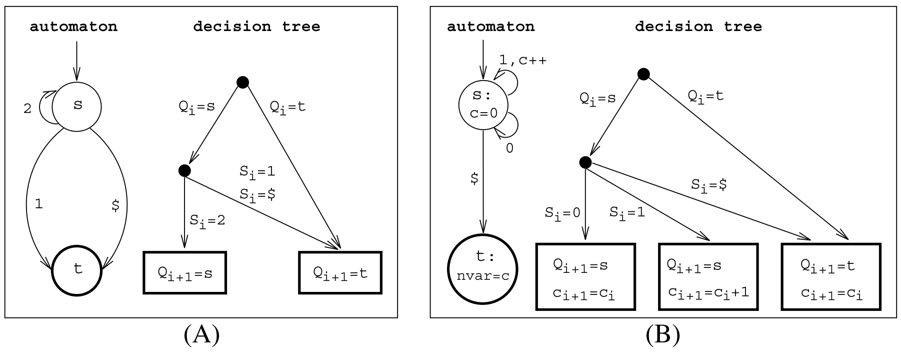


Figure 2. Automata and decision trees for (A) $\leq_{\text{lex}}$ and (B) among.

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

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

3.1. *Complete Filtering when there is No Shared Variable Between Signature Constraints*

In the general case, local consistency such as arc-consistency is not sufficient to ensure global consistency. In other words, there can be some locally consistent values that cannot be extended to a complete solution, mainly because there can be some cycles in the constraint graph. However, we will highlight some special cases where local consistency can lead to global consistency: We consider automata where all subsets of variables in *SignatureArg* are pairwise disjoint. As we will see in the section, many constraints can be encoded by such automata.

Without Counters If there are no counters, the automaton reformulation maintains arc-consistency on this kind of automata, provided that the filtering algorithms of the signature and transition constraints also maintain arc-consistency. To prove this property, consider the constraint hypergraph that represents the conjunction of all signature and transition constraints (see Figure 3). It has two particular properties: there is no cycle in

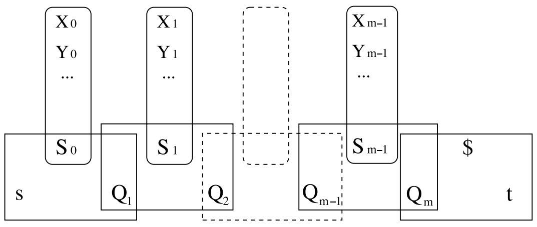


Figure 3. Constraint hypergraph of the conjunction of transition and signature constraints in the case of disjoint $\mathcal{S}ignature\mathcal{A}rg$ sets. The i-th $\mathcal{S}ignature\mathcal{A}rg$ set Δ_i is denoted by $\{X_i, Y_i, \ldots\}$.

the corresponding dual graph [19],[3] and for any pair of constraints the two sets of involved variables share at most one variable. Such a hypergraph is so-called *Berge-acyclic* [21]. Berge-acyclic constraint networks were proved to be solvable polynomially by achieving arc-consistency [22, 23]. Therefore, if all signature and transition constraints maintain arc-consistency then we obtain a complete filtering for our global constraint.

Among the 39 constraints studied in [24], eight (`between`, `between_exact-ly_one`, `global_contiguity`, `lex_different`, `lex_lesseq`, `pattern`, `two_quad_ are_in_contact`, and `two_quad_do_not_overlap`) have an automaton leading to a Berge-acyclic hypergraph.

With Counters If there are counters, some pairs of constraints share more than one variable. So, the hypergraph is not Berge-acyclic and the filtering performed may be no longer optimal. However, achieving pairwise consistency on some pairs of constraints leads to a complete filtering. To prove this, we will use the following well-know theorem in database systems.

Definition 1 [25] *A constraint network $\mathcal{N} = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ is pairwise consistent if and only if $\forall C_i \in \mathcal{C}, C_i \neq \emptyset$ and $\forall C_i, C_j \in \mathcal{C}, C_i|_s = C_j|_s$ where is the set of variables shared by C_i and C_j and $C_i|_s$ is the projection of C_i on s.*

Definition 2 [19] *An edge in the dual graph of a constraint network is redundant if its variables are shared by every edge along an alternative path between the two end points. The subgraph resulting from the removal of the redundant edges of the dual graph is called a join graph. A hypergraph has a join tree if its join graph is a tree.*

Theorem 1 [25] *A database scheme is α-acyclic* [26] *if and only if its hypergraph has a join tree.*

Theorem 2 [25] *Any pairwise consistent database over an α-acyclic database scheme is globally consistent.*

If we translate these theorems we obtain the following corollary on constraint networks.

Corollary 1 *If a constraint hypergraph has a join tree then any pairwise consistent constraint network having this constraint hypergraph is globally consistent.*

In the automata we consider here, there are no signature constraints sharing a variable. So, the dual graph of the constraint hypergraph is a tree and the hypergraph has a join tree. Therefore, the hypergraph is α-acyclic and if the constraint network is pairwise consistent, the filtering is complete for our global constraint.

A solution to reach global consistency on the network representing our global constraint is therefore to maintain pairwise consistency. In fact, if constraints share no more than one variable, pairwise consistency is nothing more than arc-consistency [23]. So, pairwise consistency has to be enforced only on pairs of constraints sharing more than one variable and only the transition constraints are therefore concerned. In the worst case, pairwise consistency will have to consider all the possible tuples of values for the set of shared variables. So, the pairs of constraints must not share too many variables if we do not want the filtering to become prohibitive.

Among the 39 constraints studied in [24], seven (`among`, `atleast`, `atmost`, `count`, `counts`, `differ_from_atleast_k_pos`, and `sliding_card_skip0`) require only one counter, two (`group`, and `group_skip_isolated_item`) need two counters and `max_index` requires three counters.

When the automaton involves only one counter, consecutive transition constraints share two variables. The sweep algorithm presented in [27] can be used to enforce pairwise consistency on each pair of transition constraints sharing two variables. Indeed, the sweep algorithm on two constraints C_i and C_j will forbid the tuples for the variables shared by C_i and C_j that cannot be extended on both C_i and C_j. To summarize, if the automaton uses one single counter, the sweep algorithm must consider each pair of transition constraints sharing variables, and arc-consistency on each constraint will lead to a complete filtering for our global constraint.

3.2. Complexity

Our approach allows the representation of a global constraint by a network of constraints of small arities. As seen in the previous section, the filtering obtained using this reformulation of the global constraint depends on the filtering performed by the solver. If the solver maintains arc-consistency using the general schema [28, 29], the complexity is in $O(er^2d^r)$[4] where e is the number of constraints, r the maximum arity and d the size of the largest domain. However, this is a rough bound and the practical time complexity can be far from this limit. Indeed, the constraints have very different arities and some domains involve only very few values. Furthermore, on some constraints a specialized algorithm can be used to reduce the filtering cost. Finally, one would want to enforce only a partial form of arc-consistency (a directional arc-consistency for example in the case of a Berge-acyclic constraint network), or a stronger filtering (enforcing pairwise consistency for example on a constraint network having a join tree). Both the pruning efficiency and the complexity of the pruning rely on the filtering performed by the solver.

3.3. *Performance*

It is reasonable to ask the question whether the automaton reformulation described herein performs anywhere near the performance delivered by a specific implementation of a given constraint. To this end, we have compared a version of the Balanced Incomplete Block Design problem [30, 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 600 MHz Pentium III. The results are shown in Table 1.

A third experiment was designed to measure the overhead of an automaton reformulation wrt. decomposing into built-in constraints. To this end, we produced random instances of the following constraints studied in [24]: `among`, `between`, `lex_lesseq`. These constraints were chosen because their automata formulations maintain the same consistency as their decomposed formulations, that is, they perform exactly the same domain filtering. Hence, comparing the time it takes to compute all solutions should give an accurate measurement of the overhead. `among` is not a built-in constraint; it can be decomposed into a number of reified membership constraints and a sum constraint. It is worth noting that its automaton uses a counter. `between`, `lex_lesseq`

Table 1. Time in milliseconds for finding (A) the first solution of BIBD instances using built in vs. simulated $\leq_{\mathrm{lex}}$ (BCS denotes time spent for breaking column symmetries: with respect to the first column, BCS corresponds to the time spent in the built-in $\leq_{\mathrm{lex}}$ constraint), and (B) all solutions to a sinlge built-in vs. simulated $\leq_{\mathrm{lex}}$ constraint

(A)

Problems v, b, r, k, λ	Built-in $\leq_{\mathrm{lex}}$ BCS/Other	Simulated $\leq_{\mathrm{lex}}$ BCS/Other
6, 50, 25, 3, 10	70/170	250/170
6, 60, 30, 3, 12	120/110	50/110
8, 14, 7, 4, 3	10/80	50/80
9, 120, 40, 3, 10	480/1090	440/1090
10, 90, 27, 3, 6	550/90	1010/90
10, 120, 36, 3, 8	1400/2070	1040/2070
12, 88, 22, 3, 4	450/970	530/970
13, 104, 24, 3, 4	540/1230	540/1230
15, 70, 14, 3, 2	220/910	520/910

(B)

m	Built-in $\leq_{\mathrm{lex}}$	Simulated $\leq_{\mathrm{lex}}$
$x_i \in [0, m-1], y_i = m - i \, \lvert\overrightarrow{x}\rvert = \lvert\overrightarrow{y}\rvert = m$		
4	10	20
5	110	170
6	1640	2300
7	29530	39100

can be expressed as built-in `lex_chain` constraints. The results are presented in three scatter plots in Figure 4. Each graph compares times for finding all solutions to a random constraint instance, using a randomly chosen labeling strategy. The X coordinate of each point is the runtime for an automaton reformulation. The Y coordinate is the runtime for

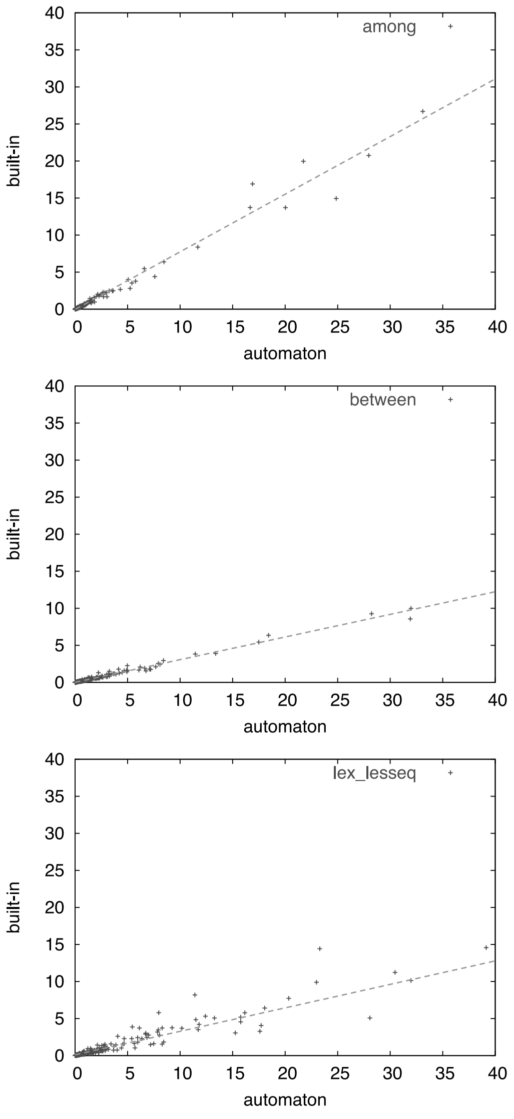


Figure 4. Scatter plots for finding all solutions to random instances: automaton reformulation vs. decomposition to built-ins.

a decomposition into built-in constraints. A line, least-square fitted to the data points, is shown in each graph. Runtimes are in seconds.

From these experiments, we observe that an automaton reformulation typically runs a few times slower than its hard-coded counterpart, roughly the same ratio as between interpreted and compiled code in a typical programming language. This slowdown is likely to have much less impact on the overall execution time of the program. The conclusion is that the automaton reformulation is a feasible and very reasonable way of rapidly prototyping new global constraints, before embarking on developing a specific filtering algorithm, should that be deemed necessary.

4. Applications of this Technique

4.1. Designing Automaton Reformulations for Global Constraints

We apply this new methodology for designing automaton reformulations for the following fairly large set of global constraints. We came up with an automaton[5] for the following constraints:

- Unary constraints specifying a domain like in [31] or `not_in` [32].
- Channeling constraints like `domain_constraint` [33].
- Counting constraints for constraining the number of occurrences of a given set of values like `among` [14], `atleast` [32], `atmost` [32] or `count` [31].
- Sliding sequence constraints like `change` [34], `longest_change` or `smooth` [15]. `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 [9] like `element_greatereq` [35], `element_lesseq` [35] or `element_sparse` [32].
- Variations around the `maximum` constraint [10] 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` [12], `group` [32], `group_skip_isolated_item` [15] or `pattern` [11].
- Constraints between vectors of variables like `between` [1], $\leq_{\text{lex}}$ [13], `lex_different` or `differ_from_at_least_k_pos`. Given two vectors $\overrightarrow{x}$ and $\overrightarrow{y}$ which have the same number of components, the constraints `lex_different` $(\overrightarrow{x}, \overrightarrow{y})$ and `differ_from_at_least_k_pos` $(k, \overrightarrow{x}, \overrightarrow{y})$ respectively enforce the vectors $\overrightarrow{x}$ and $\overrightarrow{y}$ to differ from at least 1 and k components.
- Constraints between n-dimensional boxes like `two_quad_are_in_contact` [32] or `two_quad_do_not_overlap` [36].
- Constraints on the shape of a sequence of variables like `inflexion` [15], `top` [37] or `valley` [37].
- 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* contain at least *atleast* and at most *atmost* values from the set of values *values*.

4.2. *Automaton Reformulation for a Conjunction of Global Constraints*

Another typical use of our new methodology is to come up with an automaton reformulation for the conjunction of several global constraints. This is usually difficult since it 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`$\left(\overrightarrow{a}, \overrightarrow{x}, \overrightarrow{b}\right)$ [1] and the `exactly_one`$\left(\overrightarrow{x}, values\right)$ constraints for which we come up with an automaton reformulation, which maintains arc-consistency. The `between` constraint holds iff $\overrightarrow{a} \leq_{\text{lex}} \overrightarrow{x}$ and $\overrightarrow{x} \leq_{\text{lex}} \overrightarrow{b}$, while the `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 Figure 5 depicts the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively associated with the `between` and the `exactly_one` constraints, ehile 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 conjuction of elementary conditions, where each condition is taken in one of the following set of conditions $\{a_i < x_i, a_i = x_i, a_i > x_i\}$, $\{b_i > x_i, b_i = x_i, b_i < x_i\}$, $\{x_i \in values, x_i \notin values\}$. This makes up to $3 \cdot 3 \cdot 2 = 18$ possible combinations and leads to the

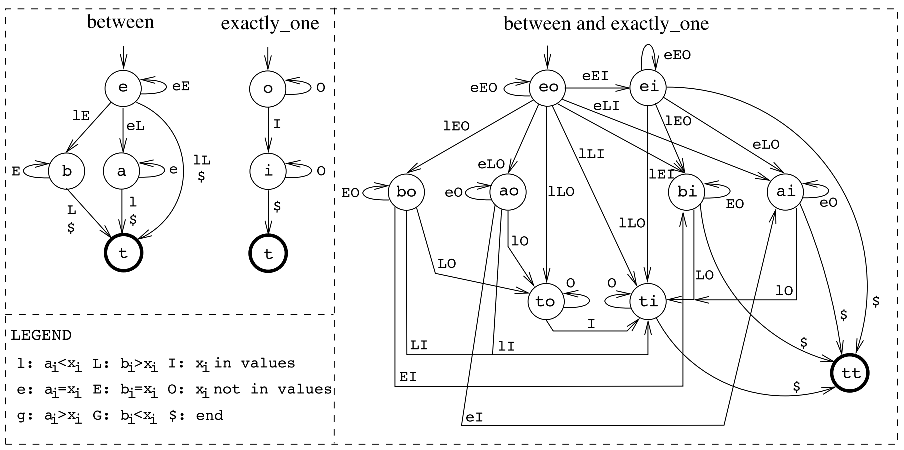


Figure 5. Automata associated with `between` and `exactly_one` and the automaton associated with their conjunction.

signature constraint $\Psi_{\texttt{between} \wedge \texttt{exactly\_one}}(S_i, a_i, x_i, b_i, values)$ between the signature variable S_i and the i-th component of vectors $\overrightarrow{a}$, $\overrightarrow{x}$ and $\overrightarrow{b}$:

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

In order to maintain arc-consistency on the conjunction of the $\texttt{between}(\overrightarrow{a}, \overrightarrow{x}, \overrightarrow{b})$ and the $\texttt{exactly\_one}(\overrightarrow{x}, values)$ 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 [31].[6]

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 maintain arc-consistency, we need the automaton associated with their conjuction to find out that $z \neq 0$. This can be seen as follows: after two transitions, the automaton $\mathcal{A}_3$ 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 supporty for $z = 0$.

4.3. Designing Constraints Between Two Sequences of Variables

This section considers constraints of the form $C(N, [VAR_1, VAR_2, \ldots, VAR_{\mathrm{m}}])$ for which the corresponding automaton $\mathcal{A}_C$ uses one single counter, which is incremented by certain transitions and finally unified to variable N in the final state of $\mathcal{A}_C$. This corresponds to constraints that count the number of occurrences of a given pattern in a sequence $VAR_1\ VAR_2 \ldots\ VAR_m$. Each time the automaton recognizes a pattern in the sequence, it increments its counter by +1. Given the automaton $\mathcal{A}_C$ of such a constraint C we will shoe how to create the automata associated to the following constraints:

– synchronized$_C$($[U_1, U_2, \ldots, U_m]$, $[V_1, V_2, \ldots, V_m]$) enforces that all positions where the automaton $\mathcal{A}_C$ recognizes a pattern in $U_1U_2 \ldots U_m$ corresponds exactly to the positions where $\mathcal{A}_C$ recognizes a pattern in $V_1\ V_2 \ldots V_m$. In other words the positions where both counters are incremented should coincide in both sequences.
– distinct$_C$($[U_1, U_2, \ldots, U_m]$, $[V_1, V_2, \ldots, V_m]$) enforces that all positions where the automaton $\mathcal{A}_C$ recognizes a pattern in $U_1U_2 \ldots U_m$ be distinct from all positions where $\mathcal{A}_C$ recogbizes a pattern in $V_1\ V_2 \ldots V_m$. This means that the positions where a counter is incremented should all be distinct.

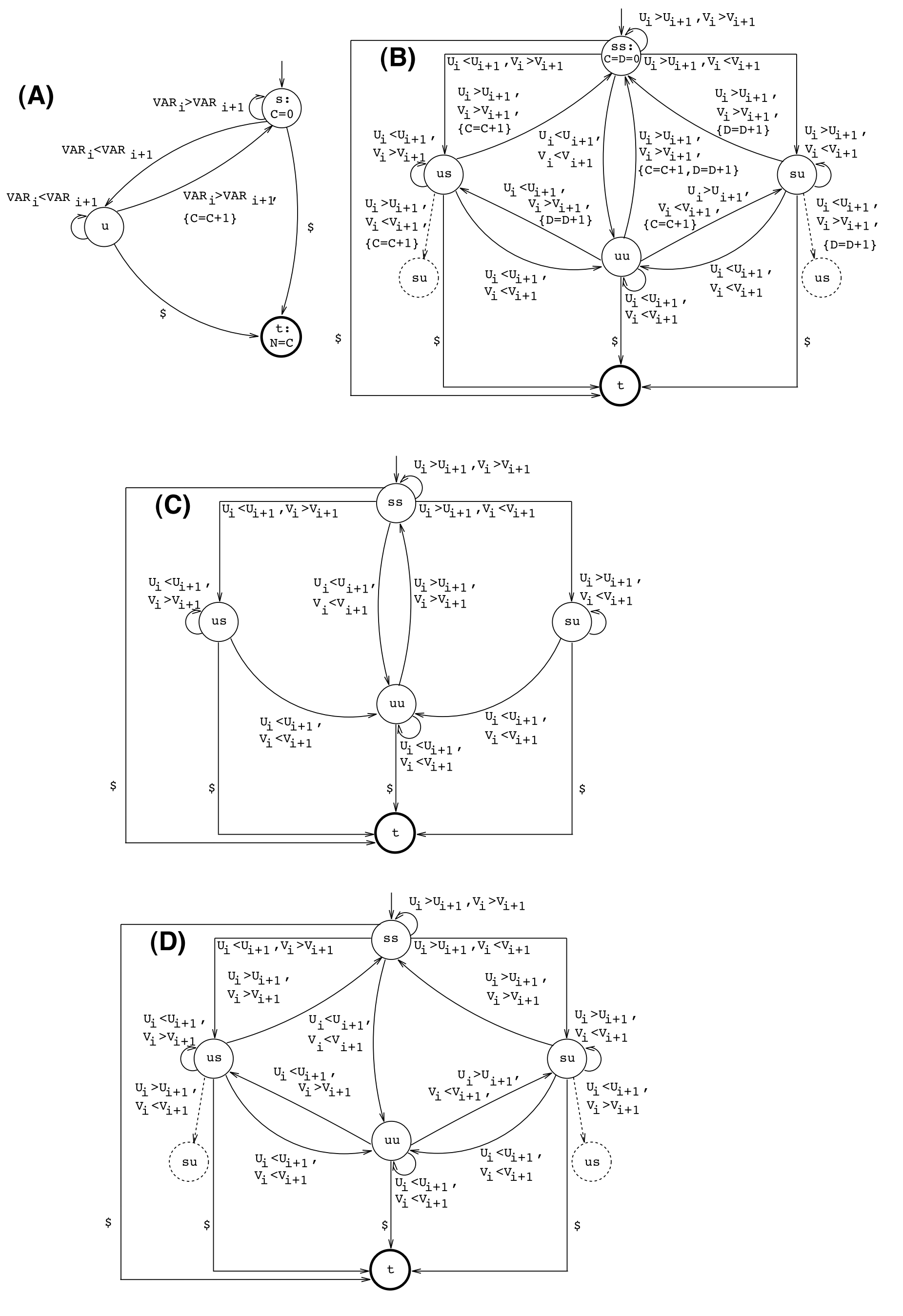


Figure 6. Automaton associated to the peak constraint and derived automata.

As an illustrative example, consider the `peak` constraint, which counts the number of peaks of a sequence of variables where adjacent variables are not equal. The automaton of this constraint is depicted by part (A) of Figure 6.

We explain how to derive the automaton associated to synchronized$_C$ and to distinct$_C$ from the automaton $\mathcal{A}_C$ associated to C. Let $\mathcal{A}_C^2$, $S(\mathcal{A}_C)$ and $D(\mathcal{A}_C)$ respectively denotes the automaton that is the product of $\mathcal{A}_C$ and $\mathcal{A}_C$, the automaton associated to synchronized$_C$, and the automaton associated to distinct$_C$. Regarding the example of the peak constraint $\mathcal{A}_C^2$ is given by part (B) of Figure 6.

We compute $S(\mathcal{A}_C)$ and $D(\mathcal{A}_C)$ from $\mathcal{A}_C^2$. Within $\mathcal{A}_C^2$ we partition its transitions into the following 4 classes:

- Those transitions that do not increment any counter,
- Those transitions that only increment the counter associated to the sequence U_1 $U_2 \ldots U_m$
- Those transitions that only increment the counter associated to the sequence V_1 $V_2 \ldots V_m$.
- Finally, those transitions that increment both counters.

$S(\mathcal{A}_C)$ corresponds to $\mathcal{A}_C^2$ from which we remove all transitions where only one single counter is incremented, and $D(\mathcal{A}_C)$ corresponds to $\mathcal{A}_C^2$ from which we remove all transitions where both counters are incremented. We also remove from the resulting automata all counter variables. Coming back to the example of the peak constraint, $S(\mathcal{A}_C)$ and $D(\mathcal{A}_C)$ respectively corresponds to part (C) and (D) of Figure 6.

5. Handling Relaxation for a Counter-Free Automaton

This section presents a filtering algorithm for handling constraint relaxing under the assumption 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 [6].

Definition 3 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.

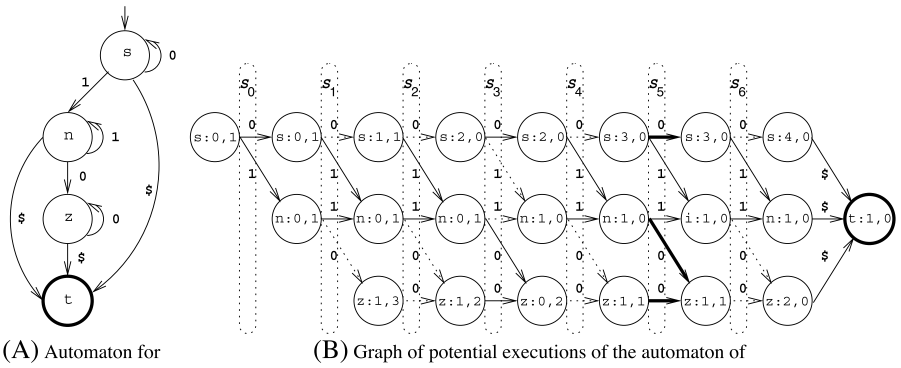
When these subsets form a partition over the variables of the constraint and when they consist of a single element, this cost is in fact the minimum number of variables to unassign in order to get to a solution. As in [7], we add a cost variable *cost* as an extra argument of the constraint. Our filtering 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 5. 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 Peasant [6], 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 valu 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, denoted by *source* and *sink* respectively.

Example 5 continued. Part (A) of Figure 7 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 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 $\mathcal{G}$. While performing the topological sort, we compute for each node n_k of $\mathcal{G}$ the minimum number



(A) Automaton for global_contiguity

(B) Graph of potential executions of the automaton of global_contiguity according to $S_0 S_1 S_2 S_3 S_4 S_5 S_6$

Figure 7. Relaxing the `global_contiguity` constraint.

of infeasible arcs from the source $\mathcal{G}$ to n_k. This number is recorded in *before*$[n_k]$.[7] 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.*

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 *after*$[n_k]$ for a node n_k of $\mathcal{G}$: *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 $\min_{a \to b \in \mathcal{A}_l^i}(before[a] + after[b])$ represents the minimum violation cost under the hypothesis that S_l remains assigned to i. If that quantity is greater than than $\mathcal{M}in$ then there is no path from *source* to *sink* that uses an arc of $\mathcal{A}_l^i$ and that 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:

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

The filtering algorithm is then based on the following theorem:

Theorem 3 *Let i be a value from the domain of a signature variable S_l. If $\mathcal{M}in_l^i >$* max(*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 5 continued. Let us come back to the instance of Figure 7. Beside the state's name, each node n_k of part (B) of Figure 7 gives the values of *before*$[n_k]$ and of *after*$[n_k]$. Since *before*$[_{sink}] = 1$ 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(*cost*) is equal to 1 we can remove value 0 from V_5.The corresponding arcs are depicted with a thick line in Figure 7.

6. Conclusion and Perspectives

The automaton description introduced in this article can 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 and the different perspectives 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 automized. 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 [38], which was the original 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 a task of executing in a non-deterministic way the deterministic program corresponding to a constraint checker and to extract the relevant information that is common to all execution paths. This can indeed be achieved by using constraint programming.
- A natural continuation would be to extend the automaton description in order to get closer to a classical imperative programming language. This would allow the direct use of available checkers in order to systematically get an automaton reformulation.
- Other structural conditions on the signature and transition constraints could be identified to guarantee arc-consistency for the original globalo constraint.
- An extension of our approach may give a systematic way to get an algorithm (not necessarily polynomial) for decision problems for which one can provide a polynomial certificate. From [39] the decision version of every problem in NP can be formulated as follows: Given x, decide whether there exists y so that $|y| \leq m(x)$ and $R(x, y)$. x is an instance of the problem; y is a short YES-certificate for this instance; $R(x, y)$ is a polynomial time decidable relation that verifies certificate y for instance x; and $m(x)$ is a computable and polynomially bounded complexity parameter that bounds the length of the certificate y. In our context, if $\left| y \right|$ is fixed and known, x is a global constraint and its $|y|$ variables with their domains; y is a solution to that global constraint; $R(x, y)$ is an automaton, which encodes a checker for that global constraint.

Acknowledgments

We are grateful to I. Katriel for suggesting the use of topological sort for the relaxation part, and to C. Bessière for his helpful comments wrt. Berge-acyclic CSP's.

Notes

1. We don't skip the checker since, in the long term, we consider that the goal would be to turn any existing code performing a check into a constraint.
2. Within the corresponding automata depicted by parts (A2), (B2), (C2) and (D2) of Figure 1, we assume that firing a transition increments from 1 the counter `i`.
3. To each constraint corresponds a node in the dual graph and if two constraints have a nonempty set S of shared variables, then there is an edge labelled S between their corresponding nodes in the dual graph. The dual graph is also called *intersection graph* in data base theory [20].
4. We assume here that the cost of a constraint check is linear in the constraint arity while it is sometimes assumed to be constant.
5. These automata are available in the technical report [24]. All signature constraints are encoded in order to maintain arc-consistency.
6. The corresponding code is available in the technical report [24].
7. The acyclic graph is layered and no transitive arcs exist. Therefore for each node n_k at a given layer k, the quantity $before[n_k]$ is computed from the nodes of layer $k - 1$ as follows: for each n_{k-1} such that there exists an arc from n_{k-1} to n_k, we add 1 to $before[n_{k-1}]$ if it is an infeasible arc, and 0 if it is a feasible arc. Then $before[n_k]$ is the minimum of such computed quantities.

References

1. Carlsson, M., & Beldiceanu, N. (2004). From constraints to finite automata to filtering algorithms. In Schmidt, D., ed., *Proc. ESOP2004*, volume 2986 of *LNCS*, pages 94–108. Springer-Verlag.
2. Vempaty, N. R. (1992). Solving constraint satisfaction problems using finite state automata. In *National Conference on Artificial Intelligence (AAAI-92)*, pages 453–458. AAAI Press.
3. Amilhastre, J. (1999). Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes. PhD thesis.
4. Amilhastre, J., Fargier, H., & Marquis P. (2002). Consistency restoration and explanations in dynamic CSPs—application to configuration. *Artif. Intell.* 135: 199–234.
5. Boigelot, B., & Wolper, P. (2002). Representing arithmetic constraints with finite automata: An overview. In Stuckey, Peter J., ed., *ICLP'2002, Int. Conf. on Logic Programming*, volume 2401 of *LNCS*, pages 1–19. Springer-Verlag.
6. Pesant, G. (2003). A regular language membership constraint for sequence of variables. In *Workshop on Modelling and Reformulation Constraint Satisfaction Problems*, pages 110–119.
7. Petit, T., Régin, J.-C., & Bessière, C. (2001). Specific filtering algorithms for over-constrained problems. In Walsh, T., ed., *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 451–463. Springer-Verlag.
8. Milano, M. (2004). *Constraint and Integer Programming*. Kluwer Academic Publishers, ISBN 1-4020-7583-9.
9. Van Hentenryck, P., & Carillon, J.-P. (1988). Generality vs. specificity: An experience with AI and OR techniques. In *National Conference on Artificial Intelligence (AAAI-88)*.
10. Beldiceanu, N. (2001). Pruning for the minimum constraint family and for the number of distinct values constraint family. In Walsh, T., ed., *CP'2001, Int. Conf. on Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 211–224. Springer-Verlag.
11. Bourdais, S., Galinier, P., & Pesant, G. (2003). HIBISCUS: A constraint programming application to staff

scheduling in health care. In Rossi, F., ed., *CP'2003, Principles and Practice of Constraint Programming*, volume 2833 of *LNCS*, pages 153–167. Springer-Verlag.

12. Maher, M. (2002). Analysis of a global contiguity constraint. In *Workshop on Rule-Based Constraint Reasoning and Programming*, held along CP-2002.
13. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., & Walsh, T. (2002). Global constraints for lexicographic orderings. In Van Hentenryck, Pascal, ed., *Principles and Practice of Constraint Programming (CP'2002)*, volume 2470 of *LNCS*, pages 93–108. Springer-Verlag.
14. Beldiceanu, N., & Contejean, E. (1994). Introducing global constraints in CHIP. *Math. Comput. Model.* 20(12): 97–123.
15. Beldiceanu, N. (2000). Global constraints as graph properties on structured network of elementary constaints of the same type. In Dechter, R., ed., *CP'2000, Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 52–66. Springer-Verlag.
16. Carlsson, M., et al. (2004) *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 3.11 edition (January) http://www.sics.se/sicstus/.
17. Le Provost, T., & Wallace, M. (1992). Domain-independent propagation. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 1004–1011. http://www.icparc.ic.ac.uk/eclipse/.
18. ILOG. *ILOG Solver Reference Manual*, 6.0 edition. http://www.ilog.com.
19. Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. *Artif. Intell.* 38: 353–366.
20. Maier, D. (1983). *The Theory of Relational Databases*. Rockville, MD: Computer Science Press.
21. Berge, C. (1970) *Graphs and Hypergraphs*. Paris: Dunod.
22. Janssen, P., & Vilarem, M.-C. (1988). Problèmes de satisfaction de contraintes: Techniques de résolution et application à la synthèse de peptides. *Research Report C.R.I.M.*, 54.
23. Jègou, P. (1991). 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.
24. Beldiceanu, N., Carlsson, M., & Petit, T. (2004). *Deriving Filtering Algorithms from Constraint Checkers*. Technical Report T2004-08, Swedish Institute of Computer Science.
25. Beeri, C., Fagin, R., Maier, D., & Yannakakis, M. (1983). On the desirability of acyclic database schemes. *JACM*, 30: 479–513.
26. Fagin, R. (1983). Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30: 514–550 (July).
27. Beldiceanu, N., & Carlsson, M. (2001). Sweep as a generic pruning technique applied to the nonoverlapping rectangles constraints. In Walsh, T., ed., *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 377–391. Springer-Verlag.
28. Bessière, C., Freuder, E. C., & Régin, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.* 107: 125–148.
29. Bessière, C., Régin, J.-C., Yap, R. H. C., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artif. Intell.* to appear.
30. Gent, I. P., & Walsh, T. (1999). CSPLib: A benchmark library for constraints. Technical Report, APES-09-1999, APES. http://www.csplib.org.
31. Carlsson, M., Ottosson, G., & Carlson, B. (1997). An open-ended finite domain constraint solver. In Glaser, H., Hartel, P., & Kuchen, H., eds., *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag.
32. COSYTEC (2003). *CHIP Reference Manual*, v5 edition.
33. Refalo, P. (2000). Linear formulation of constraint programming models and hybrid solvers. In Dechter, R., ed., *Principles and Practice of Constraint Programming (CP'2000)*, volume 1894 of *LNCS*, pages 369–383. Springer-Verlag.
34. Beldiceanu, N., & Carlsson, M. (2001). Revisiting the cardinality operator and introducing the cardinality-path constraint family. In Codognet, P., ed., *ICLP'2001, Int. Conf. on Logic Programming*, volume 2237 of *LNCS*, pages 59–73. Springer-Verlag.
35. Ottosson, G., Thorsteinsson, E., & Hooker, J. N. (1999). 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.

36. Beldiceanu, N., Guo, Q., & Thiel, S. (2001). Non-overlapping constraints between convex polytopes. In Walsh, T., ed., *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 392–407. Springer-Verlag.
37. Beldiceanu, N., & Poder, E. (2004). Cumulated profiles of minimum and maximum resource utilisation. In *Ninth Int. Conf. on Project Management and Scheduling*, pages 96–99.
38. Cohen, J. (1979). Non-deterministic algorithms. *ACM Comput. Surv.* 11(2): 79–94.
39. Woeginger, G. J. (2003). Exact algorithms for NP-hard problems: A survey. In Juenger, M., Reinelt, G., & Rinaldi, G., eds., *Combinatorial Optimization—Eureka! You shrink!*, volume 2570 of *LNCS*, pages 185–207. Springer-Verlag.