DFA-based formulation for constraint negation ⋆

Nadjib Lazaar[1], Noureddine Aribi[3], Arnaud Gotlieb[2], Yahia Lebbah[3]

[1] INRIA-Microsoft Research Joint Centre, 28 rue Jean Rostand 91893 Orsay Cedex, France
nadjib.lazaar@inria.fr
[2] Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
arnaud@simula.no
[3] Laboratoire LITIO, Université d'Oran, ,B.P. 1524 EL-M'Naouar, 31000 Oran, Algérie
{ylebbah, aribi.noureddine}@gmail.com

Abstract. Global constraint design is a key success of CP for solving hard combinatorial problems. Many works suggest that automaton-based definitions and filtering make easier the design of new global constraints. In this paper, from such a design, we present a preliminary work that gives an automaton-based definition of the NEGATION of a global constraint. For a given global constraint C, the idea lies in giving operators for computing an automaton that recognizes only tuples that are not solution of C, and use the REGULAR global constraint to automatically reason on this automaton. We implemented this approach for automaton-based global constraints, including `global_contiguity` and $\leq_{\texttt{lex}}$ constraints, and got experimental results that show that their automatically computed negation is highly competitive with more syntactic transformations.

1 Introduction

Modern constraint programming languages aim at making easy problems formulation and solving. One of the key success of CP is global constraints design. Since its introduction in [6], automaton-based definition of global constraint has grown and is now recognized as a mainstream technique. Carlsson and Beldiceanu proposed in [6,3] to use automata representation and reformulation for designing new global constraints from constraint checkers. Pesant proposed in [14] a generic global constraint, the REGULAR global constraint which holds if a fixed-length sequence of finite-domain variables represents a word of a given regular language. In another context, Andersen et al. [1] proposed to use the multivalued decision diagram structure (MDD) to replace the domain store where constraints have an MDD-Based presentation.

As suggested by the above mentioned works, building new global constraints is often required to address challenging combinatorial problems. Obviously, having *logical negation* in the tool-box would be interesting to facilitate this process. In our previous works related to program verification [10,11,12], we faced the problem of negating existing global constraints. Our (naive) solution involved simple syntactic transformations of the original constraints to easily compute its negation. For example, the negation of: `inverse(all[R](i in R) g[i], all[S](j in S) f[j]);` in OPL was easily expressed by:

⋆ This work is supported by INRIA-DGRSDT (France, Algeria).

$\texttt{or(i in S) g[f[i]]! = i; or(j in R) f[g[j]]! = j.}$
As possible, the syntactic transformations can exploit also the existing global constraints to express the negation form of a given constraint. For example, the negation of an `atLeast` constraint can be expressed using the `atMost` and vice-versa, `GCC` by `atLeast` and `atMost`, `allDifferent` by a disjunction of `GCC`, etc.

However, those syntactic transformations did not capture the essence of *logical negation* and did not filter constraints in a sufficient and consistent way.

In (Constraint) Logic Programming, *negation–as–failure* has been the traditional approach to deal with negation in the general framework of the Clark completion. However, it is well known that *negation–as–failure* corresponds only to logical negation on ground instances. *Constructive negation*, as proposed by Stuckey in [18], presents a sound and complete operational model of negation in the Herbrand Universe. An interesting implementation of this operator in the constraint concurrency model of Oz has been proposed by Schulte in [16]. Constructive constraint negation is general as it can handle any constraint but is also ineffective in terms of filtering. Indeed, no dedicated filtering algorithms is available for the negation of the constraint and thus, these operators are usually not useful to prune the search space. More recently, constraint negation has been considered in the more general context of *logical connectives* [5,2,13]. However, in these works, negation is proposed for constraints defined in extension and cannot be applied to global constraints that capture complex relations among a set of variables.

In this paper, we present a preliminary work that takes the automaton-based design of a global constraint as input and automatically returns an automaton-based definition of the NEGATION of this global constraint. For a given global constraint C, the idea is first to give operators for computing a Deterministic Finite Automaton (DFA) that recognizes the tuples that are not solution of C ; and second to use the REGULAR global constraint [14] to automatically derive filtering rules for this new automaton.

One can choose an MDD-based design and just swap end-states to get the negation form. But this approach has two limitations: First, it is expensive, because for a given constraint, the generated MDD contains only the feasible paths. To do such negation, the infeasible portion has to be generated as well as the feasible one. Second, for efficiency reasons, MDD-based global constraints are usually represented by fixed-width MDDs [8]. This representation may include assignments violating the constraint, thus, building the negation of a given global constraint by swapping the end-states between accepting and non-accepting states in a fixed-width MDD, may be unsound. On the contrary, we will see that using a folded DFA for building the negation is guaranteed to be sound.

This paper contains global constraint examples that were automatically negated through our approach, including the negation of the `global_contiguity` and $\leq_{\texttt{lex}}$ constraints. We implemented our approach in Gecode, where a good implementation of REGULAR is available, and got some experimental results on these global constraints that show our negation is competitive with more syntactic transformations.

The paper is organized as follows. The next section describes the process of constructing the automaton of the negated constraint and using REGULAR. Section 3 illustrates the approach on two constraints: `global_contiguity`, and $\leq_{\texttt{lex}}$. The experimentations are described in section 4. Section 5 concludes the paper.

2 Negation on DFA-based Global Constraints

In this section, we present an efficient method to handle the negation of the automaton-based design global constraints. This kind of global constraint has behind a specific DFA (Deterministic Finite Automaton) as a checker of ground instances. We summarize the approach in two points:
- From the DFA of a given global constraint C, we generate, using an automatic process, the complement which is the DFA of the negation form ($\neg$C).
- We derive the filtering algorithm using the REGULAR constraint.

2.1 Notations

A Deterministic Finite Automata (DFA) $\mathcal{A}$ of a given constraint C is defined as a 7-tuple, $(X, E, \Psi, \Sigma, \delta, e_0, F)$, consisting of:

- a sequence of finite-domain variables X (i.e., signature of the constraint C),
- a finite set E of states e_i ,
- a finite set of labeled states Ψ s.t. $source(e_0)$: the starting state, $node(e_i)$: intermediate state, $sink(e_i)$: sink state[4], $final(e_i)$: final state,
- a finite alphabet Σ,
- a transition function δ, (e_i, s, e_j) is a transition where $e_i, e_j \in E, s \in \Sigma \cup \{\$\}$[5],
- the start state e_0 where $source(e_0) \in \Psi$,
- a set of final states $F \subseteq E$ where $\forall e_i \in F : final(e_i) \in \Psi$.

We stress the fact that any state e_i should have exclusively one of the four possible labels in Ψ. We add the empty symbol \$ as a transition in order to be complete by treating the end of the parsed word. Moreover, it simplifies the proposed process of negating the automata. We note $\mathcal{L}$ the language recognized by $\mathcal{A}$ (i.e., $\mathcal{L}(\mathcal{A})$) where $\mathcal{L} \subseteq \Sigma^*$.

2.2 DFA Complement

To have the complement of a global constraint DFA's we use three operators, namely: *Complete*, *Swap-state* and *Clean-up* operators.

A Complete DFA A deterministic automaton $\mathcal{A}$ of a given constraint C considers essentially the patterns where the constraint is evaluated to the accepted or satisfied states (i.e., *final* states). It does not consider all the possible patterns of the constraint. The patterns that violate the constraint are not considered, because they do not lead to satisfaction states. Thus, to complete an automaton we have to add all possible states in order to be able to consider all the possible patterns of the constraint instance (i.e., $\mathcal{C}(\mathcal{A})$). Formally speaking, the *complete* operator on $\mathcal{A}$ returns an extended automaton $\mathcal{C}(\mathcal{A})$ that takes in account all transitions as well as those leading to a *sink* state.

[4] State e_i is a *sink* state if and only if it is not a *final* state and there are no transitions leading from e_i to another state.

[5] \$ represents the empty transition.

Definition 1 ($\mathcal{C}(\mathcal{A})$) *The complete automaton of $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ is $\mathcal{C}(\mathcal{A}) = (X, E', \Psi', \Sigma, \delta', e_0, F)$ s.t.:*

- $E' = E \cup \{e_k : \exists s \in \Sigma, e_i \in E \; s.t. \; (e_i, s, e_k) \notin \delta\}$
- $\Psi' = \Psi \cup \{sink(e_k) : e_k \in E' \backslash E\}$
- $\delta' = \delta \cup \{(e_i, s, e_k) : \exists s \in \Sigma, e_i \in E', e_k \notin E\}$

For $|E| = n$ and $|\Sigma| = m$, the *complete* operator adds at most nm states and transitions, and is $\mathcal{O}(nm)$. It is correct where it preserves $\mathcal{L}(\mathcal{A})$ by adding only *sink* states. It is also complete where, for each state of the computed DFA's, there is as outgoing arcs as symbols in Σ.

A DFA Swap-state The *swap-state* $\mathcal{S}$ swaps the *sink* states to *final* and vice-versa.

Definition 2 ($\mathcal{S}(\mathcal{A})$) *Let us take the automaton $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$, a swap-state on $\mathcal{A}$ is $\mathcal{S}(\mathcal{A}) = (X, E, \psi', \Sigma, \delta, e_0, F')$ s.t.:*

$$\forall e_i, e_j \in E : \; final(e_i), sink(e_j) \in \Psi \Rightarrow sink(e_i), final(e_j) \in \Psi'$$

It is obvious to say that the *swape-state* operator is correct and complete where all (and only) *sink* (resp. *final*) states are swapped. The first step of completing the automaton is essential to get quickly the negated form through the current second swapping step. If we swap first, we should take into account that the final state of the negated form should be added, which is the job done by the first step.

A DFA Clean-up The *Clean-up* operator on a DFA $\mathcal{A}$, noted $\mathcal{U}(\mathcal{A})$, is the inverse function of the *complete* operator (i.e., $\mathcal{U}(\mathcal{C}(\mathcal{A})) = \mathcal{C}(\mathcal{U}(\mathcal{A})) = \mathcal{A}$) , where the *clean-up* reduces the automaton by removing all transitions leading to a *sink* state.

Definition 3 ($\mathcal{U}(\mathcal{A})$) *The clean-up operator of $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ is $\mathcal{U}(\mathcal{A}) = (X, E', \Psi', \Sigma, \delta', e_0, F)$ s.t.:*

- $E' = E \backslash \{e_k : sink(e_k) \in \Psi\}$
- $\Psi' = \Psi \backslash \{sink(e_k) : e_k \in E\}$
- $\delta' = \delta \setminus \{(e_i, s, e_k) \in E \times \Sigma \times E \; : \; sink(e_k) \in \Psi\}$

The *clean-up* operator preserves $\mathcal{L}(\mathcal{A})$ where it cannot remove a *final* or a *node* state (*correctness*). At the end, the computed DFA's removes all *sink* states and transitions leading to its (*completeness*).

Using the three operators seen before, we get the complement of a given DFA of a global constraint.

Proposition 1 *Let $\mathcal{A}$ a DFA. $\bar{\mathcal{A}}$ is the complement s.t.:*

$$\bar{\mathcal{A}} = \mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$$

PROOF Let $\mathcal{L}(\mathcal{A})$ (resp. $\mathcal{L}(\mathcal{B})$) a regular language for some DFA $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ (resp. $\mathcal{B} = (X, E', \Psi', \Sigma, \delta', e_0, F')$) s.t. $\mathcal{B} = \mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$. $\mathcal{B}$ is the complement of $\mathcal{A}$ iff $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ (i.e., $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^* - \mathcal{L}(\mathcal{A})$ as stated by [9]).

- $w \in \mathcal{L}(\mathcal{A}) \Rightarrow w \notin \mathcal{L}(\mathcal{B})$: Let $w \in \mathcal{L}(\mathcal{A})$, so $\exists e_i$ s.t.$(e_0, w, e_i) \in \delta^*$ and $final(e_i) \in \Psi$. e_i is a *final* state in $\mathcal{C}(\mathcal{A})$ (**Def.1**) where it is swapped to *sink* state by $\mathcal{S}(\mathcal{C}(\mathcal{A}))$ (**Def.2**). By $\mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$ e_i will be removed as it is *sink* state, so $final(e_i) \notin \Psi'$ (**Def.3**) and $w \notin \mathcal{L}(\mathcal{B})$.
- $w \in \mathcal{L}(\mathcal{B}) \Rightarrow w \notin \mathcal{L}(\mathcal{A})$: In the same way, the inverse is also true.

Property 1 *The complement of a regular language is regular [9].*

The property guarantees that the complement of a given DFA automaton $\mathcal{A}$ (i.e., the recognized regular language $\mathcal{L}$) is a DFA (i.e., regular language).

Property 2 *The complement of a* DFA *of a given constraint* C *represents the* DFA *of the negated form* $\neg$C.

A DFA of a given constraint C represents the solution set of the constraint, therefore all instantiations that do not belong to this solution set are recognized by the complement DFA. So, the complement DFA represents the solution set of $\neg$C.

2.3 Filtering the negation with the REGULAR constraint

Having the automaton is not enough to get a filtering algorithm of the negation of a given global constraint: rules associated to the regular expressions recognized by the automaton have to be considered [6]. While automatic construction of the automaton of the negation is easy, finding filtering rules is difficult, especially when generalized arc-consistency is required. In the general case, as the automaton for the negation is a DFA that can be augmented with counters, the generic global constraint GRAMMAR could be used to automatically derive filtering rules [17,15]. However, this constraint has exponential cost w.r.t. the states of the automaton. When strings are of fixed length, [7] pointed out an approach where the GRAMMAR constraint is processed with the REGULAR global constraint [14] by transforming the push-down automaton associated to the constrained grammar to a finite-state automaton. Thus, in our approach, we selected the REGULAR global constraint to encode generic filtering rules for the negation of global constraints.

A *regular language membership constraint* is a constraint C on a sequence of finite-domain variables x associated with a DFA $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ s.t:

$$regular(\mathtt{x}, \mathcal{A}) = \{\tau \ : \ \tau \ tuple \ of \ \mathtt{x} \ recognized \ by \ \mathcal{A}\}$$

The consistency algorithm of the REGULAR constraint has three main phases. The *forward*, the *backward* and the *maintaining* phases collect states from E that support the

pair (x_i, v_i) (i.e., $v_i \in D_{x_i}$).

The *forward* phase unfolds the DFA $\mathcal{A}$ by constructing the corresponding Multivalued Decision Diagram (MDD) which is an acyclic graph by construction. The MDD contains different layers L_i ($L_1, L_2, \ldots, L_n$). Each layer contains states from E where arcs appear between consecutive layers. The first layer L_1 contains only the start state e_0 ($source(e_0) \in \Psi$). We unfold the DFA from L_1 to L_n according to the transition function σ.

The *backward* phase removes states and the corresponding incoming arcs from layer L_n to L_1. We start by removing from the last Layer L_n all no *final* states and their incoming arcs. For a layer L_i, we remove all states and their incoming arcs that have no outgoing arcs.

There is also a *maintaining* phase if a domain reduction is provoked by another constraint. Here the MDD needs to be maintained by removing all arcs corresponding to the removed value. We remove also, for each layer, all unreachable states or those without outgoing arcs.

To show how REGULAR is used in our framework, we illustrate the automatic derivation for the negation of `global_contiguity` and `Lex` in the next section.

3 Case Studies

In this section we take two case studies to illustrate our approach, namely `global_contiguity` and $\leq_{\texttt{lex}}$ constraints. We construct DFA of the negated form and we get the filtering algorithm using the REGULAR constraint.

3.1 Case study: ¬`global_contiguity`

The `global_contiguity(var)` [3,4] is defined on a vector of variables `var`. Each variable `var[i]` can take value in $\{0, 1\}$. The `global_contiguity` constraint holds since the valuation of the sequence of variables `var` contains no more than one group of contiguous 1. For example, if we take a sequence of 10 variables, the sequence 0011110000 is a correct sequence where 0011100110 is not.

The DFA of `global_contiguity` constraint is given in Fig. 1 **part(a)**. It corresponds to:
$\mathcal{A} = (var, \{e_0, e_1, e_2, e_3\}, \Psi, \{0, 1\}, \delta, e_0, \{e_3\})$,
$\Psi = \{source(e_0), node(e_1), node(e_2), final(e_3)\}$,
$\delta = \{(e_0, 0, e_0), (e_0, 1, e_1), (e_0, \$, e_3), (e_1, 1, e_1), (e_1, 0, e_2),$
$(e_1, \$, e_3), (e_2, 0, e_2), (e_2, \$, e_3)\}$.

To construct the complement of the DFA shown in Fig.1 **part(a)**, we first call the *complete* operator (Fig.1 **part(b)**) where the *sink* state e_4 is added to complete the automaton. Second, the *swap-states* operator swaps the added state e_4 to *final* and the *final*

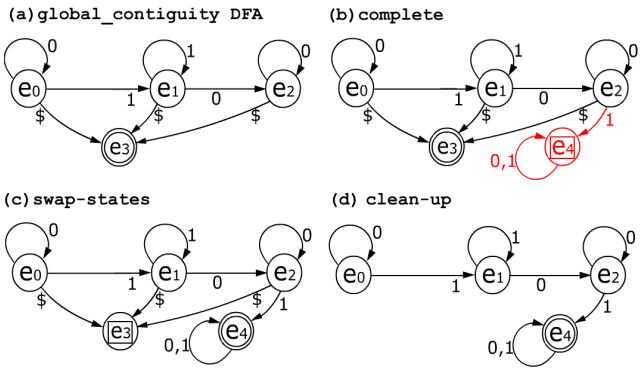

Fig. 1. Complement DFA of the `global_contiguity` constraint.

state e_3 is swapped to *sink* state (Fig.1 **part(c)**). The *clean-up* step removes the resulting *sink* state e_3 (Fig.1 **part(d)**).

Once the DFA of the negated form constructed, we exploit the filtering algorithm of the REGULAR constraint.

The regular expression of the consistent tuples of `global_contiguity` is given by:

$$0^*1^*0^*$$

If we consider the negation form, we have as regular expression of the consistent tuples of ¬`global_contiguity`:

$$0^*11^*00^*1\{0,1\}^*$$

These two regular expressions can be easily modeled in any CP language containing the REGULAR constraint.

The fact that the automaton of `global_contiguity` constraint is defined on the variables values, enabled to exploit efficiently and directly the REGULAR constraint.

Let us take an example with four variables (x_1, x_2, x_3, x_4). Fig.2 shows the three phases of REGULAR consistency algorithm. The MDD is constructed with four layers corresponding to the variables. The *forward* phase unfolds the negated DFA of Fig.1 where the *backward* phase removes 9 arcs and 6 states. If another constraint reduces the domain of x_3 by removing the value 0, the *maintaining* phase removes 6 arcs and 3 states to have at the end only two solutions (1010 and 1011).

3.2 Case study: ¬Lex

The lexicographic ordering constraint[3,4] $\boldsymbol{x} \leq_{lex} \boldsymbol{y}$ over two vectors of variables $\boldsymbol{x} =< x_0, x_1, ..., x_{n-1} >$ and $\boldsymbol{y} =< y_0, y_1, ..., y_{n-1} >$ holds iff $n = 0$, or $x_0 < y_0$, or $x_0 = y_0$, and $\boldsymbol{x} =< x_1, ..., x_{n-1} > \leq_{lex} < y_1, ..., y_{n-1} >$. The automaton is defined on the relation between every two consecutive variables. In order to exploit the REGULAR constraint, we should transform the Lex constraint as following:

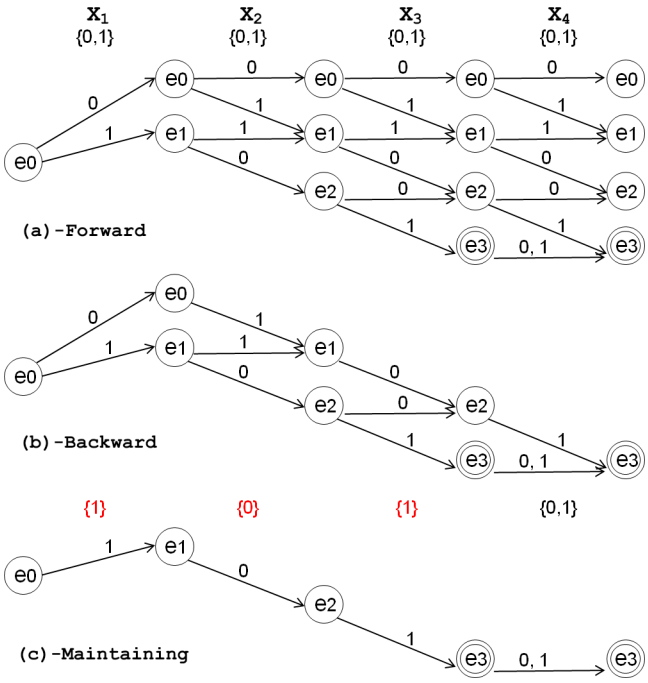

Fig. 2. REGULAR constraint on ¬`global_contiguity` with four variables.

$$\boldsymbol{x} \leq_{lex} \boldsymbol{y} \equiv LexRel(r_0, r_1, ..., r_{n-1})$$

where $r_i \in \{<, =, >\}$, and $rel(r_i, x_i, y_i) \equiv x_i \; r_i \; y_i$. The automaton of $LexRel(r_0, r_1, ..., r_{n-1})$ is given in Figure 3 **part(a)** where it corresponds to:
$\mathcal{A} = ((\boldsymbol{x}, \boldsymbol{y}), \{e_0, e_1, e_2\}, \Psi, \{=, <, >\}, \delta, e_0, \{e_1, e_2\})$,
$\Psi = \{source(e_0), final(e_1), final(e_2)\}$,
$\delta = \{(e_0, =, e_0), (e_0, <, e_1), (e_0, \$, e_2), (e_1, =, e_1), (e_1, <, e_1), (e_1, >, e_1)\}$.

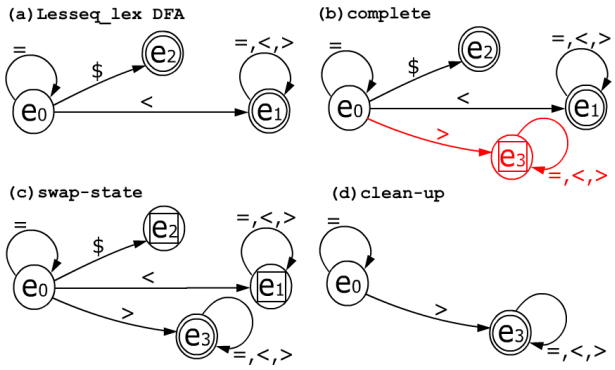

Fig. 3. Complement DFA of the $\leq_{\texttt{lex}}$ constraint.

This constraint can be easily implemented with the REGULAR constraint, where the accepted regular expression is:

$$=^* \mid =^* < \{=, <, >\}^*$$

Negating form of the $\leq_{\texttt{lex}}$ constraint is shown in **part(d)** of the Figure 3 which represents the complement of the DFA of $\leq_{\texttt{lex}}$. **part (b,c,d)** shows respectively the *complete*, *swap-state* and *clean-up* steps to obtain the complement. From the complement of this constraint (i.e., DFA of $\neg\leq_{\texttt{lex}}$) we get the associated regular expression:

$$=^* > \{=, <, >\}^*$$

With the regular expression of the negated $\leq_{\texttt{lex}}$, we are able to exploit efficiently and directly the REGULAR constraint for filtering.

Let us take a simple example with $\boldsymbol{x} = [x_1, x_2, x_3, x_4]$ and $\boldsymbol{y} = [y_1, y_2, y_3, y_4]$ where each variable takes a value in $[0, 10]$. Fig.4 shows the three phases of REGULAR consistency algorithm on $\neg \leq_{lex}$. The MDD is constructed with four layers corresponding to the variables (x_i, y_i). The *forward* phase unfolds the negated DFA of Fig.3 and the *backward* phase removes the state e_0 form the last layer. In the case of a reduction by other constraints where the domain of x_1 is reduced to $[5, 10]$ and y_1 to $[0, 4]$, the *maintaining* phase removes 6 arcs and 3 states.

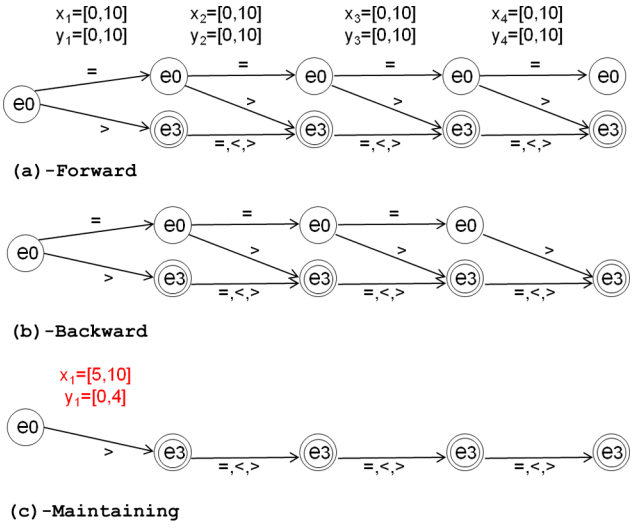

Fig. 4. REGULAR constraint on $\neg \leq_{\texttt{lex}}$ with four variables.

4 Experimental validation

The goal of our experimental validation was to check that an automaton-based negated global constraint version (gotten for free through the presented framework) was more effective than a version where the negation is syntactically computed. The negated syntactic form can be a global constraint, as the case of the $>_{lex}$ for $\leq_{lex}$. But usually the negated syntactic form is obtained by decomposing the global constraint into primitive constraints where the negation can be done on the logical combination of these primitive constraints. In this decomposition case, we associate to each primitive constraint a logical variable, and then the negation is done by negating the logical form on the new logical variables.
For both the `global_contiguity` and $\leq_{\texttt{lex}}$ constraints, we built Gecode models and run our experiments on Intel Core2Quad CPU, Q6600 of 2.4 GHz, Linux machine with 3 Go of RAM.

4.1 `global_contiguity`

The declarative specification of `global_contiguity` can be given by:

$$\texttt{global\_contiguity}(\texttt{x}) \equiv \forall i, j \in 1..n : i < j \; s.t.$$

$$(x_i = 1) \wedge (x_j = 0) \Rightarrow (\forall k \in j + 1..n : x_k = 0)$$

Where, the negated form can be declaratively given by:

$$\neg\texttt{global\_contiguity}(\texttt{x}) \equiv \exists i, j \in 1..n : i < j \; s.t.$$

$$(x_i = 1) \wedge (x_j = 0) \wedge (\exists k \in j+1..n : x_k = 1)$$

The Table 1 contains our experimental results on `global_contiguity`. We give a comparison between the implementation of the declarative specification of ¬`global_contiguity` and DFA-based implementation using our negation approach and the REGULAR constraint. The reported results are on different instances (from 200 to 11.10^3 variables) where a solving to get the first 100 solutions is launched. The results are on time/memory consumptions, number of propagations and the generated nodes. For each instance, from 200 to 10^3, the DFA-Based negation gives an interesting and impressive results comparing to the syntactic transformations based negation. For example, let us take the instance of 10^3 variables, the syntactic approach take more than five minutes and $2.5\ Go$ of memory. With our DFA approach, the solving to get the first 100 solutions takes only $32\ ms$ and $9\ Mo$ of memory. For big instances (more than 10^3 variables), the syntactic approach reports an out-of-memory. Our approach stills giving interesting results also for the huge instance (11.10^3 variables) with $32\ sec.$. Fig. 5 shows the increase of time consumption of a solving to get the first 100 solutions with a syntactic negation and a DFA-based negation according to the grow-up of instances. The time consumption in the syntactic transformations increase following an exponential, where the increase time using our approach is in a linear way.

Table 1. Experimental results on ¬`global_contiguity`.

var	syntactic transformations based negation				DFA − based negation			
	T	M	P	N	T	M	P	N
200	53.13	24.39	65 111	396	2.09	0.42	366	397
300	186.07	77.90	116 535	496	3.31	0.77	468	497
400	509.93	179.22	170 209	596	4.81	1.559	566	597
500	1 082.35	344.86	244 235	696	6.69	2.20	668	697
600	1 936.68	589.79	315 320	796	8.77	2.96	766	797
700	3 125.01	930.34	411 935	896	11.43	4.80	868	897
800	4 735.71	1 381.68	500 407	996	14.19	6.03	966	997
900	6 760.44	1 960.24	619 627	1 096	17.27	7.28	1 068	1 097
1 000	19 407.02	2 681.01	725 507	1196	22.88	8.53	1 166	1 197
1 100	—	OOM	—	—	24.17	9 925	1 268	1 297
1 200	—	OOM	—	—	28.67	14 214	1 366	1 397
1 300	—	OOM	—	—	32.78	16 330	1 468	1 497
1 400	—	OOM	—	—	37.58	18 766	1 566	1 597
1 500	—	OOM	—	—	42.69	21 266	1 668	1 697
1 600	—	OOM	—	—	47.83	23 702	1 766	1 797
1 700	—	OOM	—	—	53.34	26 276	1 868	1 897
1 800	—	OOM	—	—	59.32	28 712	1 966	1 997
1 900	—	OOM	—	—	65.09	31 212	2 068	2 097
11 000	—	OOM	—	—	1 923.53	896 958	11 166	11 197

T : time(ms), M : memory(MB), P : propagations, N : nodes, OOM : Out − Of − Memory

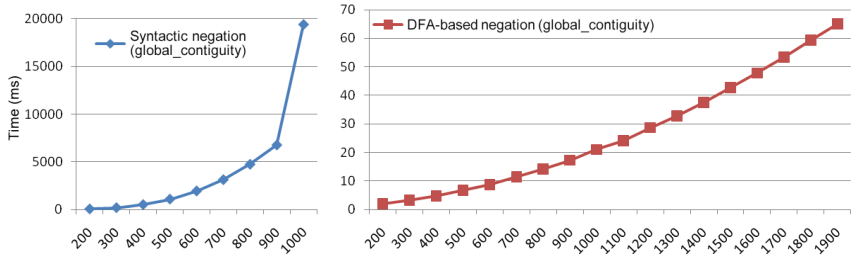

Fig. 5. Time consumptions for ¬`global_contiguity` (syntactic and DFA-Based negation).

4.2 $\leq_{lex}$

The declarative specification of $\leq_{lex}$ on $\boldsymbol{x} =< x_0, x_1, ..., x_{n-1} >$ and $\boldsymbol{y} =< y_0, y_1, ..., y_{n-1} >$ can be given as follows:

$$\boldsymbol{x} \leq_{lex} \boldsymbol{y} \equiv (n = 0) \vee (x_0 < y_0) \vee$$

$$(x_0 = y_0 \wedge < x_1, ..., x_{n-1} > \ \leq_{\texttt{lex}} \ < y_1, ..., y_{n-1} >)$$

where the negation form is simply:

$$\neg(\boldsymbol{x} \leq_{lex} \boldsymbol{y}) \equiv ((n = 1) \wedge (x_0 > y_0)) \vee ((n > 1) \wedge ((x_0 > y_0) \vee$$
$$(< x_1, ..., x_{n-1} > \ \neg\leq_{\texttt{lex}} \ < y_1, ..., y_{n-1} >)))$$

This is a first transformation to get the negation of $\leq_{\texttt{lex}}$. One can also express the negation using the global constraint $>_{\texttt{lex}}$ which is available on Gecode:

$$\neg(\boldsymbol{x} \leq_{lex} \boldsymbol{y}) \equiv \boldsymbol{x} >_{lex} \boldsymbol{y}$$

The Table 2 contains results on the syntactic negation and the $>_{\texttt{lex}}$ global constraint. We compare the two results with our DFA-based negation approach. The reported results are on 200 to 8.10^3 variables instances. The DFA-based negation is better than the syntactic negation and the original constraint $>_{\texttt{lex}}$. Let us take the big instance with 8.10^3 variables, our generic approach to negate $\leq_{\texttt{lex}}$ have a time consumption three times less than the syntactic negation and two times less than $>_{\texttt{lex}}$ constraint. For memory consumption, the $>_{\texttt{lex}}$ have a consumption two times or more than the DFA-based negation. Through these comparisons, we see that the DFA-based negation is widely better than the syntactic negation and remains very competitive with its equivalent well established global constraint $>_{lex}$.

5 Conclusion

In this paper, we have proposed a preliminary approach to get automatically a filtering algorithm for the negation of an automaton-based global constraint. This approach

Table 2. Experimental results on $\neg \leq_{\texttt{lex}}$.

var	syntactic transformations				$>_{\texttt{lex}}$				DFA			
	T	M	P	N	T	M	P	N	T	M	P	N
200	7.00	1.67	1 944	400	6.60	2.03	2 341	400	4.27	0.82	302	400
300	13.24	3.50	2 708	500	12.07	4.69	3 311	500	7.07	2.10	402	500
400	21.45	6.42	3 544	600	19.09	7.64	4 341	600	11.16	3.00	502	600
500	30.86	9.56	4 308	700	27.65	11.87	5 311	700	15.28	4.32	602	700
600	43.32	13.35	5 144	800	38.13	16.03	6341	800	20.30	7.75	702	800
700	56.77	17.38	5 908	900	49.67	20.90	7 311	900	26.28	9.00	802	900
800	71.89	22.00	6 744	1 000	62.72	26.73	8 341	1 000	32.62	11.41	902	1 000
900	90.12	27.12	7 508	1 100	77.12	33.59	9 311	1 100	39.70	15.03	1 002	1 100
10^3	107.97	33.02	8 344	1 200	92.54	40.50	10 341	1200	47.38	16.57	1 102	1 200
2.10^3	402.09	123.06	16 344	2 200	334.72	153.91	20 341	2 200	161.96	65.46	2 102	2 200
3.10^3	889.68	270.51	24 344	3 200	731.38	352.69	30 341	3 200	344.10	147.19	3 102	3 200
4.10^3	1 591.25	475.89	32 344	4 200	1 300.39	625.27	40 341	4 200	597.54	255.35	4 102	4 200
5.10^3	2 527.08	738.53	40 344	5 200	2 059.30	970.92	50 341	5 200	915.20	395.56	5 102	5 200
6.10^3	3 758.49	1 059.63	48 344	6 200	3 010.67	1 388.53	60 341	6 200	1 310.84	567.67	6 102	6 200
7.10^3	5 194.56	1 440.67	56 344	7 200	4 115.96	1 879.97	70 341	7 200	1 772.84	770.78	7 102	7 200
8.10^3	6 951.67	1 880.27	64 344	8 200	5 681.13	2 446.10	80 341	8 200	2 309.79	1 004.37	8 102	8 200

T : time(ms), M : memory(MB), P : propagations, N : nodes, OOM : Out − Of − Memory

is built over automata operations and exploits the REGULAR global constraints to automatically derive filtering rules for the negation. Through experiments, we evaluated this approach on two global constraints (i.e., `global_contiguity` and $\leq_{\texttt{lex}}$), using Gecode, where results are encouraging. We forecast 1) to extend our approach to push-down automata by using the generic GRAMMAR constraint and 2) to extend it to logical connectives (i.e., conjunction, disjunction) between global constraints.

References

1. H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pages 118–132, Berlin, Heidelberg, 2007. Springer-Verlag.
2. Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.
3. Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.
4. Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12:21–62, March 2007.
5. Jefferson C., Moore N. C. A., Nightingale P., and Petrie K. E. Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429, 2010.
6. Mats Carlsson and Nicolas Beldiceanu. From constraints to finite automata to filtering algorithms. In *ESOP*, pages 94–108, 2004.
7. Katsirelos G., Narodytska N., and Walsh T. Reformulating global grammar constraints. In *CPAIOR*, volume 5547 of *LNCS*, pages 132–147. Springer, 2009.
8. Samid Hoda, Willem-Jan Van Hoeve, and J. N. Hooker. A systematic approach to mdd-based constraint programming. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 266–280, Berlin, Heidelberg, 2010. Springer-Verlag.

9. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.
10. Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. Fault localization in constraint programs. In *ICTAI (1)*, pages 61–67. IEEE Computer Society, 2010.
11. Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. On testing constraint programs. In *CP*, volume 6308 of *LNCS*, pages 330–344. Springer, 2010.
12. Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A framework for the automatic correction of constraint programs. In *ICST*, page Forthcomming. IEEE Computer Society, 2011.
13. Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR 2004*, volume 3011 of *LNCS*, pages 209–224, 2004.
14. Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *CP*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.
15. Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 751–755. Springer, 2006.
16. Christian Schulte. Programming deep concurrent constraint combinators. In *PADL 2000*, volume 1753 of *LNCS*, pages 215–229. Springer, 2000.
17. Meinolf Sellmann. The theory of grammar constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 530–544. Springer, 2006.
18. Peter J. Stuckey and Peter J. Stuckey. Negation and constraint logic programming, 1995.