Reifying Global Constraints

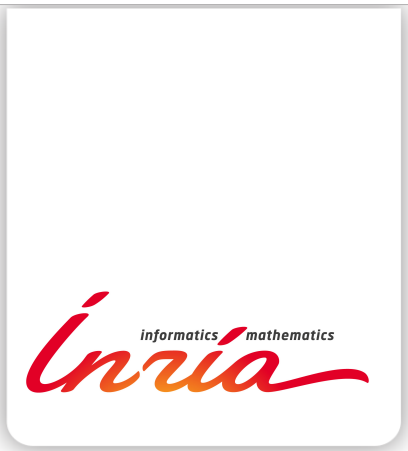
Francois Fages, Sylvain Soliman

Reifying Global Constraints

François Fages, Sylvain Soliman

RESEARCH REPORT
N° 8084
October 2012
Project-Team Contraintes

ISSN 0249-6399 ISRN INRIA/RR--8084--FR+ENG



Reifying Global Constraints

François Fages, Sylvain Soliman

Project-Team Contraintes

Research Report n° 8084 — October 2012 — 18 pages

Abstract: Global constraints were introduced two decades ago as a means to model some core aspects of combinatorial problems with one single constraint for which an efficient domain filtering algorithm can be provided, possibly using a complete change of representation. However, global constraints are just constraint schemas on which one would like to apply usual constraint operations such as reification, i.e. checking entailment, disentailment and negating the constraint. This is currently not the case in state-of-the-art tools and was not considered in the global constraint catalog until recently. In this paper, we propose a general framework for reifying global constraints and apply it to some important constraints of the catalog, such as the cumulative constraint for instance. We show that several global constraints that were believed to be hard to negate can in fact be efficiently negated, and that entailment and disentailment can be efficiently tested. We also point out some new global constraints that are worth studying from this point of view and provide some performance figures obtained with an implementation in Choco.

Key-words: Global constraints, reification, modelling

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Réifier les contraintes globales

Résumé : Les contraintes globales ont été introduites il y a une vingtaine d'années afin de modéliser certains aspects centraux des problèmes combinatoires avec une seule contrainte dotée d'un algorithme de filtrage efficace, au besoin via un changement complet de représentation. Cependant, les contraintes globales ne sont que des schmas de contraintes sur lesquelles on souhaiterait pouvoir appliquer les opérations usuelles des contraintes comme la réification, ce qui suppose de tester l'implication et de nier la contrainte. Ceci n'est pas le cas dans les outils de l'état de l'art et n'a été considéré que récemment dans le catalogue des contraintes globales. Dans cet article nous proposons un cadre général pour réifier les contraintes globales, et l'appliquons aux principales contraintes du catalogue, comme par exemple la contrainte cumulative. Nous montrons que plusieurs contraintes réputées difficiles à nier peuvent l'être efficacement, et que l'implication peuvent tre teste efficacement. Nous montrons aussi que de nouvelles contraintes globales vaudraient la peine d'être étudiées de ce point de vue, et fournissons une évaluation préliminaire des performances obtenues avec une imprémentation en Choco.

Mots-clés : Contraintes globales, réification, modélisation

1 Introduction

Reified constraints have proved useful for modeling many combinatorial problems. The reification of a constraint c is a constraint, noted $B \leftrightarrow c$, which contains one extra Boolean variable B that is true if and only if the constraint c is true. Reified constraints allow us to apply logical connectives to constraints and make it possible to define higher-order constraints such as cardinality constraints. For example, by summing up the zero-one variables of a list of reified constraints, one can define a cardinality constraint which can bound the number of constraints that are true in a list of constraints.

Beyond that, global constraints were introduced two decades ago as a means to model some core aspects of combinatorial problems with one single constraint for which efficient domain filtering algorithms can be provided, possibly using a complete change of representation. However, global constraints are just constraint schemas on which one would like to apply usual constraint operations such as reification, i.e. checking entailment and negating the constraint. This is currently not the case in state-of-the-art tools and has not been considered in the global constraint catalog [1, 2] until recently with the notion of functional dependencies [4]. Currently, the global constraint catalog contains 364 constraints among which only one is directly reified (namely the `in_interval_reified` constraint). In addition, the so-called *reified automaton constraint* provides a means to mechanically construct a reified constraint for a constraint c from the finite deterministic automaton that only accepts the set of solutions of c. However, it has been recently shown in [4] that many global constraints can be reified through their decomposition using functional dependencies.

The reason for that situation is that the development of global constraints has been mainly pushed by applications, and there is no clear need for using reified global constraints in practical applications. On the other hand, this need appears when considering front-end high-level modeling languages such as OPL [24, 19], Zinc [20, 12], Essence [15], Rules2CP [13], since first-order logical expressions involving constraints and logical connectives have to be compiled in expressions of the constraint solver, and these complex expressions should in principle be applicable to global constraints as well.

It has also been argued that reifying global constraints seems very difficult for most constraints, e.g. for the cumulative constraint. For these reasons, [14] defines a half reification scheme which proceeds by flattening and alleviates the need for negating the constraint. Similarly in [4], the authors consider functional dependencies in global constraints and present a simple method for negating constraints based on these decompositions.

In this paper we show that there no reason to restrict ourselves to these particular cases, and that many global constraints can in fact be efficiently reified using a general reification scheme. In particular, we show that the cumulative constraint can be efficiently reified.

In the next section, we provide general definitions for a global constraint in any computation domain, discrete or continuous, and for the domain consistency and domain entailment notions associated to domain filtering and entailment detection algorithms respectively. We then describe our general reification scheme for global constraints and its implementation in Choco [23] for finite domains. Then in Section 3, we present its application to several important constraints of the catalog of global constraints over finite domains, and discuss in each

case the level of consistency achieved for the negation of the constraint and the completeness of the entailment conditions. In Section 4, we present some performance figures obtained with our implementation in Choco. In Section 5, we compare our results with the related work of [14] and [4]. Finally we conclude on the generality of the scheme and on the remaining difficulties for some global constraints.

2 Reification Framework for Global Constraints

2.1 Preliminary Definitions on Global Constraints

From a logical point of view, a constraint formula is just a first-order logic formula containing free variables (plus possibly quantifiers) which is interpreted in a fixed computation domain as a mathematical relation over the free variables. In this view, a constraint is a schema of constraint formulae.

Definition 1 *A* constraint formula *is any first-order logical formula over some vocabulary Σ and interpreted in some fixed mathematical structure domain $\mathcal{D}$.*

A constraint *is a schema of constraint formulae, i.e. a family of constraint formulae obtained from some formula pattern by variable substitutions.*

A global constraint *is a constraint with an* unbounded number of free variables.

For instance, the formula $x \neq y \wedge y \neq z \wedge x \neq z$, abbreviated as *alldifferent([x,y,z])*, is a constraint formula over variables x, y, z. The schema $\bigwedge_{i<j} x_i \neq x_j$ of constraint formulae with free variables $\{x_i\}_{1 \leq i \leq n}$ for any number n of variables is the global constraint named *alldifferent*. Another example of global constraint is the family of linear equalities over any number of variables, while the family of linear equalities over three variables is one constraint that is not global according to the previous definition.

From a constraint programming and more precisely constraint propagation point of view however, a constraint is not just a pattern of logical formulae but should come with an efficient *domain filtering algorithm* which

1. maintains a domain $d \subseteq \mathcal{D}$ for each variable x,
2. reduces the domain of the free variables without losing solutions when it is not instantiated,
3. and decides the truth of the constraint when it is instantiated.

A *constraint checking algorithm* is an algorithm which does not affect the domain of the variables and performs step 3 only.

In order to measure the level of domain reductions performed by a domain filtering algorithm, and to compare different domain filtering algorithms for the same constraint, the following notions of consistency are classically considered. Let us denote by $dom(x)$ the domain of a variable x, and for those ordered domains which admit minimum and maximum elements, by $\underline{x}$ (resp. $\overline{x}$) the minimum (resp. maximum) value of $dom(x)$. The list $x_1 \ldots x_n$ will be denoted by $[x_i]$.

Definition 2 *A constraint formula c is* domain-consistent *(also called hyperarc-consistent or generalized arc-consistent) if it admits a solution for each value in the domain of its variables:*

$$\forall v_1 \in dom(x_1) \exists v_2 \in dom(x_2) \dots \exists v_n \in dom(x_n)\ \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow c$$

for all $x_1 \in V(c)$, where $\{x_2, \dots, x_n\} = V(c) \setminus \{x_1\}$.

A domain filtering algorithm is complete *if it maintains the constraint domain-consistent by removing values from the domain of the variables for which the constraint has no solution.*

For domains with minimum and maximum elements, a constraint formula c is said hull-consistent *if each bound of its variables admits a solution in the domain of the variables:*

$$\forall v_1 \in \{\underline{x_1}, \overline{x_1}\} \exists v_2 \in dom(x_2) \dots \exists v_n \in dom(x_n)\ \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow c$$

for all $x_1 \in V(c)$, where $\{x_2, \dots, x_n\} = V(c) \setminus \{x_1\}$.

A constraint formula c is bound-consistent *if each bound of its variables admits a solution in the interval domain of the variables:*

$$\forall v_1 \in \{\underline{x_1}, \overline{x_1}\} \exists v_2 \in [\underline{x_2}, \overline{x_2}] \dots \exists v_n \in [\underline{x_n}, \overline{x_n}]\ \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow c$$

for all $x_1 \in V(c)$, where $\{x_2, \dots, x_n\} = V(c) \setminus \{x_1\}$.

In a finite domain $\mathcal{D}$, one can always achieve domain-consistency in time $O(|\mathcal{D}|^n)$ but we are interested in domain filtering algorithms of low (amortized) complexity such as $O(1)$, $O(n)$, or $O(n^2)$. Bound consistency provides a weaker notion of consistency for measuring the domain reductions performed by such more efficient yet incomplete domain filtering algorithms. Hull consistency provides an intermediate notion of consistency which is mainly used for domain filtering algorithms over the reals [8]. If the domains are intervals,both notions of hull and bound consistency are equivalent.

For a global constraint, the idea is also that the level of consistency achieved by the domain filtering algorithm should be higher than the one achieved by decomposing the constraint into simpler constraints [10]. It should be clear however that such a notion of non-decomposability cannot be taken in the definition of a global constraint, since nothing prevents the discovery of clever decompositions achieving domain-consistency, as recently done for instance in [11] for the alldifferent constraint.

Definition 3 *A constraint is* domain-entailed *if*

$$\forall v_1 \in dom(x_1) \forall v_2 \in dom(x_2) \dots \forall v_n \in dom(x_n)\ \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow c,$$

and domain-disentailed *if $\forall v_1 \in dom(x_1) \dots \forall v_n \in dom(x_n)\ \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow \neg c$.*

An entailment (resp. disentailment) checking algorithm is domain-complete *if it tests the domain-entailment (resp. disentailment) condition.*

Here again, the obvious algorithm for checking domain-entailment in a finite domain is in time $O(|\mathcal{D}|^n)$ but we are interested in algorithms with constant or linear time (amortized) complexity, possibly to the expense of the loss of domain-completeness of the entailment test, e.g. by testing the stronger interval-entailment condition:

$$\forall v_1 \in [\underline{x_1}, \overline{x_1}] \forall v_2 \in [\underline{x_2}, \overline{x_2}] \ldots \forall v_n \in [\underline{x_n}, \overline{x_n}] \; \mathcal{D} \models (\bigwedge_{i=1}^{n} x_i = v_i) \Rightarrow c.$$

2.2 Reification Framework

Let us assume, without loss of generality, that the domain $\mathcal{D}$ contains two distinguished values, such as $\{0, 1\}$, for encoding false and true, and let us call a Boolean variable a variable with initial domain $\{0, 1\}$.

From a logical point of view, the *reification* of a constraint c is a constraint, written $B \leftrightarrow c$, that contains one extra Boolean variable B which is true if and only if the constraint c is true, i.e. $\mathcal{D} \models (B \leftrightarrow c) \Leftrightarrow ((B = 1 \wedge c) \vee (B = 0 \wedge \neg c))$.

From a constraint programming point of view, the reification of a constraint c consists in defining:

1. the *negation* of c, i.e. a constraint over the *same free variables*, written $\overline{c}$, such that for any ground valuation ρ, $\overline{c}\rho$ is true if and only if $c\rho$ is false, i.e. $\mathcal{D} \models \overline{c} \Leftrightarrow \neg c$;

2. an *entailment condition*, written $c^>$, over the *domains* of the free variables of c, such that $\mathcal{D} \models c^> \Rightarrow c$;

3. a *disentailment condition* $c^<$ over the domains of the free variables of c, such that $\mathcal{D} \models c^< \Rightarrow \neg c$.

Formally, the entailment conditions are logical formulae on the domains of the variables. They are used to check the variable domains without performing any domain reduction. The minimum entailment condition that can always be used consists in waiting that the constraint is fully instantiated and checking the truth of the constraint when it is fully instantiated. In practice, we are interested in *sufficient conditions* on the domains of the variables for testing with a low computational complexity whether a constraint is domain-entailed, domain-disentailed or none.

The entailment condition for the negation of the constraint, $\overline{c}^>$, is clearly a valid disentailment condition for c and can thus be chosen as definition for $c^<$. However, for the sake of generality and efficiency of the implementation, the choice of $\overline{c}^>$ is not enforced for $c^<$.

Our reification scheme can be summarized with the following pseudo-code:

```
class Constraint
   boolean entailed()
      // default entailment check for any constraint
      return instantiated() and satisfied()

class ReifiedConstraint
   void init(BoolVar b, Constraint c, Constraint not_c,
             Function entailment_check,
```

```
            Function disentailment_check)
      this.b = b
      this.c = c
      this.not_c = not_c
      if entailment_check is not undefined
         this.c_entailed = entailment_check
      else
         this.c_entailed = c.entailed
      if disentailment_check is not undefined
         this.c_disentailed = disentailment_check
      else
         this.c_disentailed = not_c.entailed

   void wake()
      if this.b is true
         post(this.c)
      if this.b is false
         post(this.not_c)
      if this.c_entailed() is true
         instantiate this.b to true
      if this.c_disentailed() is true
         instantiate this.b to false
```

2.3 Watched Literals for Entailment Conditions

It is worth noticing that the entailment conditions need not be checked at each modification of the domains of the variables and can be postponed to some particular domain modifications, similarly to what is done for constraint propagators through the use of *watched literals* [16].

For instance, in the *alldifferent*($[x_i]$) constraint, the entailment condition $\forall i < j\ dom(x_i) \cap dom(x_j) = \emptyset$ is domain-complete. If it does not hold because some value k is in the domain of two variables x_i and x_j, then the two literals $x_i = k$ and $x_j = k$ are watched literals and the entailment condition need not be tested again, as long as the particular value k is not removed from the domains of either x_i or x_j. The modifications of the domain of the other variables of the global constraint can thus be ignored for checking the entailment condition.

2.4 Implementation in Choco

The framework we have presented so far can be applied to any constraint domain (including continuous domain) but from now on, and for the rest of this article, we will focus on finite domains, i.e. we assume that $\mathcal{D} = \mathbb{N}$ and every variable is given initially with a bounded domain.

Choco [23] is a constraint programming system implemented in Java which provides several global constraints and a general scheme for defining new constraints over finite domains. We use that feature for reifying global constraints, that is for implementing reified global constraints as new constraints in the Choco system.

In this implementation, the entailment conditions are implemented with the general mechanism of events used for reified constraints, i.e., through the single

`isEntailed` method that returns `true` if the constraint is entailed, `false` if it is disentailed and `null` otherwise.

3 Application to the Global Constraint Catalog

Let us now consider some important constraints of the global constraint catalog [1, 2] mentioned in [22]. For each constraint, we provide a table of constraints and entailment conditions that can be used to reify it. We also recall the usual consistency achieved efficiently for the global constraints and analyze the completeness of the entailment and disentailment conditions.

3.1 alldifferent

The constraint *alldifferent* is the formula schema $\bigwedge_{i<j,i,j=1}^{n} x_i \neq x_j$ where n is the number of variables. Many efficient propagators have been described for it, and domain-consistency can be achieved efficiently [21].

The negation of alldifferent can naturally be defined as the schema $\exists i \exists j\ 1 \leq i < j \leq n\ x_i = x_j$ which is equivalent to $|\{x_i\}_{i=1}^{n}| < n$. It is thus possible to take as propagator of the *non-alldifferent* constraint, that of the *atmost_nvalue* global constraint, with an upper bound set to the number of variables minus one, for which bound consistency can be achieved efficiently [5].

As for entailment, one can provide a *domain-complete* entailment condition for *alldifferent* with the formula $\forall i \neq j\ dom(x_i) \cap dom(x_j) = \emptyset$. Indeed, if there is any non-empty intersection, it can be extended to a valuation showing that the constraint is not entailed by the domains. The time complexity of this entailment condition is in $O(n^2 \cdot d)$, where d is the size of the domain $\mathcal{D}$. Our previous remark on watched literals [16] in Section 2.3 plainly applies here, since that entailment condition has to be tested only when one particular value common to the domains of only two variables of the constraint is deleted.

One also has to devise a disentailment condition for *alldifferent*, i.e. an entailment condition for its negation. Interestingly,the max-cardinality-matching constraint does provide a complete domain-disentailment condition for alldifferent. However, in our reification scheme, the disentailment checker is a component separated from the constraint propagator, and the communication of internal information through the API of a constraint propagator to external components is a well-known open issue, especially considering that the max-cardinality-matching propagator should not be launched, but only used to give back the value of the cardinality of the matching. Therefore it makes sense to provide disentailment conditions that are easy to implement without having to modify constraint propagators for tasks they have not been designed for. One option is to implement an external max-cardinality-matching algorithm, *à la* Hopcroft-Karp. This algorithm implements the domain-complete condition $\bigvee_{I \subseteq \{1,\ldots,n\}} |\bigcup_{i \in I} dom(x_i)| < |I|$ with a time complexity in $O(n \cdot d \cdot \sqrt{n+d})$. Another option is to implement a non domain-complete condition using a heuristics on the choice of the set I to consider, e.g. small cardinality of I, small domain sizes for variables in I, etc. In the following table that summarizes the reification of alldifferent, we indicate the disentailment condition used in our implementation with all the sets I of cardinality 1 and n which runs in time $O(n^2 + n.d)$.

Reification	Formula and constraint propagators	Performance
posting	$\bigwedge_{i<j} x_i \neq x_j$	
	$alldifferent([x_i])$	AC [21]
negating	$\exists i < j \ x_i = x_j$	
	$atmost\_nvalue(n-1, [x_i]_{i=1}^n)$	BC [5]
entailment	$\forall i < j \ dom(x_i) \cap dom(x_j) = \emptyset$	domain-complete
	$disjoint([dom(x_i)]_{i=1}^n)$	$O(n^2 \cdot d)$
disentailment	$\lvert max\_cardinality\_matching([x_i], [dom(x_i)])\rvert < n$	domain-complete
		$O(n \cdot d \cdot \sqrt{n+d})$
	$\bigvee_{i<j} x_i = x_j \vee \lvert\bigcup_{i=1}^n dom(x_i)\rvert < n$	$O(n^2 + n.d)$

3.2 cumulative

The *cumulative* constraint defined over a list of tasks with variables o_i, d_i and h_i representing respectively the origin (start), duration and height of the task and with limit L is the schema

$$\forall i \sum_{j,\ o_j \leq o_i < o_j + d_j} h_j \leq L$$

The *non-cumulative* constraint actually amounts to saying that at some point the sum of heights of involved tasks is over the limit L. This can be encoded in a *cumulatives* constraint [3], with a single machine m with **lower bound 0**, and the same tasks (origin, duration, height) as in the original *cumulative* constraint, plus a new task of duration 1, of unconstrained origin x and of height $-(\mathbf{L}+\mathbf{1})$. Indeed, this constraint will be satisfied precisely when there is (at least) one possible value for the origin of the negative task, i.e., a point in time when the total resources used by the original tasks counterbalance the negative one and thus is above L.

Entailment can be checked by evaluating the condition
$\forall i \forall v \in dom(o_i) \ \sum_j \underline{o_j} \leq v < \overline{o_j} + \overline{d_j} \ \overline{h_j} \leq \underline{L}$.
i.e.
$\forall v \in \bigcup_i dom(o_i) \ \sum_j \underline{o_j} \leq v < \overline{o_j} + \overline{d_j} \ \overline{h_j} \leq \underline{L}$.
which is domain-complete. Similarly, a domain-complete disentailment condition is
$\exists i \forall v \in dom(o_i) \ \sum_j \overline{o_j} \leq v < \underline{o_j} + \underline{d_j} \ \underline{h_j} > \overline{L}$.
These conditions are equivalent to the formulations checking all values of v, but since the height cannot increase at any other point in time than the start of a task, it is enough to check those values of v only.

The following table summarizes our reification of the cumulative constraint.

Reification	Formula and constraint propagators	Performance
posting	$\forall i \ \sum_{j,\ o_j \leq o_i < o_j + d_j} h_j \leq L$ $cumulative([(o_i, d_i, h_i)], L)$	
negating	$\exists i \ \sum_{j,\ o_j \leq o_i < o_j + d_j} h_j > L$ $cumulatives([(x, 1, -(L+1))\|(o_i, d_i, h_i)], (m, 0, \geq))$	
entailment	$\forall v \in \bigcup_i dom(o_i) \ \sum_j \ \underline{o_j} \leq v < \overline{o_j} + \overline{d_j} \ \overline{h_j} \leq \underline{L}.$	domain-complete $O(n \cdot d)$
disentailment	$\exists i \forall v \in dom(o_i) \ \sum_j \ \overline{o_j} \leq v < \underline{o_j} + \underline{d_j} \ \underline{h_j} > \overline{L}$	domain-complete $O(n^2 \cdot d)$

It is worth remarking that, as is done by default in Choco, and contrary to the Global Constraint Catalog, we have expressed the cumulative constraint without variables for end times. Adding extra variables for end times in the definition of the cumulative constraint amounts to adding equality constraints between the origin variables plus duration and the end time variables to the semantics of the constraint. These extra constraints may indeed be better propagated within the constraint rather than outside. However, with that semantics, the negation of the constraint involves a disjunction for violating the end time constraints in addition to the negation of the core cumulative constraint. This disjunction can be propagated with constructive disjunction, by trying the violation of the end-time constraints first in order to post the negation of the core constraint, and the entailment checks can be similarly adapted.

3.3 permutation/same

The *permutation* constraint states that two lists of variables take values such that the lists are permutations of each other.

The negation of the constraint can be enforced through two *global_cardinality* constraints (3.5) each defining as many variables as the union of all domains: $global\_cardinality([x_i], [(val_i, v_i)]) \wedge global\_cardinality([y_i], [(val_i, w_i)])$ and then enforcing at least one difference with $lex\_different([v_i], [w_i])$.

As for entailment, this is an example of constraint for which the only way to verify domain-entailment is to wait until all variables are fully instantiated. This case is not very frequent but is a consequence of the limits of the representation of variables only by their domains (contrary, for instance, to models incorporating unification).

As for disentailment, there is again, as in Section 3.1, a check based on max-cardinality matching in a bipartite graph with arcs from $[x_i]$ to the corresponding values in $\mathcal{D}$ and arcs from possible values in $\mathcal{D}$ to $[y_j]$. If the max-cardinality is below n then the constraint is disentailed.

Reification	Formula and constraint propagators	Performance
posting	$\forall i \ \lvert\{j \mid x_j = i\}\rvert = \lvert\{j \mid y_j = i\}\rvert$ $permutation([x_i], [y_i])$	AC [6]
negating	$\exists i \ \lvert\{j \mid x_j = i\}\rvert \neq \lvert\{j \mid y_j = i\}\rvert$ $global\_cardinality([x_i], [(val_i, v_i)])$ $\wedge global\_cardinality([y_i], [(val_i, w_i)])$ $\wedge lex\_different([v_i], [w_i])$	
entailment	$is\_instantiated \wedge is\_satisfied$	domain-complete $O(n)$
disentailment	$\lvert max\_cardinality\_matching([x_i], [dom(x_i)] \cup [dom(y_j)], [y_j])\rvert < n$	domain-complete $O(n \cdot d \cdot \sqrt{2n + d})$

3.4 diff-n

The *diff-n* constraint on k-dimensional orthotopes, each being of the form $\langle (o_i^1, s_i^1), \ldots, (o_i^k, s_i^k) \rangle$, where the o_i^l variable denotes the origin of object i in dimension l and s_i^l its size, imposes that they do not overlap, i.e., that there be for each pair of orthotopes at least one dimension where the projections do not overlap. The version presented in the following table assumes that objects are indeed k-dimensional, i.e., $s_j^l > 0$.

Reification	Formula and constraint propagators	Performance
posting	$\bigwedge_{i<j} \bigvee_d (o_i^l \geq o_j^l + s_j^l) \vee (o_j^l \geq o_i^l + s_j^l)$	
negating	$\bigvee_{i<j} \bigwedge_d (o_i^l < o_j^l + s_j^l) \wedge (o_j^l < o_i^l + s_j^l)$	
entailment	$\bigwedge_{i<j} \bigvee_d (\underline{o_i^l} \geq \overline{o_j^l} + \overline{s_j^l})$ $\vee (\underline{o_j^l} \geq \overline{o_i^l} + \overline{s_j^l})$	domain-complete $O(n^2 \cdot k)$
disentailment	$\bigvee_{i<j} \bigwedge_d (\overline{o_i^l} < \underline{o_j^l} + \underline{s_j^l})$ $\wedge (\overline{o_j^l} < \underline{o_i^l} + \underline{s_j^l})$	domain-complete $O(n^2 \cdot k)$

Domain-complete entailment and disentailment conditions are easily obtained for this constraint through simple bounds reasoning. Negating the constraint, i.e. propagating the *overlap* constraint, is more problematical. It is worth remarking that once again, like for the cumulative constraint, eliminating the end variables from the core constraint avoids that the negation involves objects with inconsistent dimensions. Nevertheless here, the negated constraint is a big disjunction for which no efficient global propagator was found.

Although not motivated by practical application, but for the sake of reifying the diff-n constraint, the *overlap* constraint thus appears as a global constraint worth studying in its own right.

3.5 global cardinality

The *global_cardinality* constraint $global\_cardinality([x_i], [(val_j, m_j)])$ imposes that each value val_j appears exactly as many times in the list of variables $[x_i]$ as the value of variable m_j.

Since the $m_j, 1 \leq j \leq k$ are variables, this is a special case of the *open* variant of the global cardinality constraint [25].

The negation is obtained through another cardinality constraint on fresh variables m'_i and the added constraint that the vectors $[m_i]$ and $[m'_i]$ differ at at least one place.

Once again, domain entailment is only possible once all variables have been instantiated. As in Section 3.3, the disentailment condition relies on compulsory and possible values of variables, it can be seen as a flow problem in the graph described in [25], but it is not domain-complete since it only considers the bounds $\overline{m_j}$ and $\underline{m_j}$ (as capacity and demand from the values to the unique target vertex).

Reification	Formula and constraint propagators	Completeness
posting	$global\_cardinality([x_i], [(val_j, m_j)])$	
negating	$global\_cardinality([x_i], [(val_j, m'_j)])$ $\wedge lex\_different([m_j], [m'_j])$	
entailment	$is\_instantiated \wedge is\_satisfied$	domain-complete $O(n)$
disentailment	$integer\_flow([x_i], [(val_j, \overline{m_j}, \underline{m_j})])$	$O((n+k)^3)$

3.6 increasing

The *increasing* constraint imposes that a list of variables take increasing values, i.e. that they are sorted in increasing order. The entailment and disentailment conditions are straightforward. The only notable point is that the *change* global constraint can be used as negation, by counting the number of elements

Reification	Formula and constraint propagators	Performance
posting	$increasing([x_i])$	AC
negating	$change(c, [x_i], >) \wedge c \geq 1$	AC [7, 18]
entailment	$\bigwedge \overline{x_i} \leq \underline{x_{i+1}}$	domain-complete $O(n)$
disentailment	$\bigvee \underline{x_i} > \overline{x_{i+1}}$	domain-complete $O(n)$

3.7 k-diff/atleast_nvalue

The $atleast\_nvalue(m, [x_i])$ constraint imposes that the list of variables $[x_i]$ takes at least m different values. Similarly to alldifferent, the max-cardinality-matching computed by the propagator, it can be used for a domain-complete disentailment check.

Reification	Formula and constraint propagators	Performance
posting	$atleast\_nvalue(m, [x_i])$	AC [9]
negating	$atmost\_nvalue(m-1, [x_i])$	BC [5]
entailment	$\lvert\{dom(x_i) \mid \forall j \neq i \ dom(x_i) \cap dom(x_j) = \emptyset\}\rvert \geq m$	
disentailment	cf. $alldifferent$	domain-complete

4 Experimental Results

In order to provide a minimum evaluation of the power and overhead of our reification scheme for global constraints, one can consider some standard benchmark involving global constraints, and double the global constraint with a reified version of it. We measure the overhead in computation time for solving the problem with a reified vs non-reified constraint posted in parallel.

The constraint tested is *alldifferent*, for which two benchmarks were chosen: *contrived* (size 200-200), which has no solutions and thus involves many backtracks and constraint awakenings and *langford* for which we chose on the contrary an instance with solutions (size 2-100). See [17] for details[1]. We compare the execution time (for 10 repetitions) as given by Choco `runtimeStatistics` for four variants: the global constraint alone, and the global constraint in conjunction with a reified version (i.e., $alldifferent \wedge (B \leftrightarrow alldifferent)$) for three possible implementations of the entailment/disentailment conditions. Note that the negation remains the same each time (the global constraint *atmost_nvalue*).

As described in Section 3.1 there is a whole family of available entailment and disentailment conditions. We present here the results for:

- a minimal one, that waits until full instantiation to check for satisfaction;
- one that checks on top of that for disentailment by using for I the n sets $\{1\}, \{1,2\}, \ldots, \{1,2,\ldots,n\}$;
- one that adds also entailment by domain disjunction, as implied by their bounds;
- a final one, similar to the previous one, but using watched literals as described in Section 2.3.

As shown in Table 1, the overhead of implementing the general scheme, i.e., adding a negation and minimal entailment/disentailment conditions, is quite

[1] see also `http://minion.sourceforge.net/benchmarks.html` "Benchmarks for alldifferent"

Problem	No reif.	Minimal reif.	Disentailment	Full reification	Watched lit.
contrived	42.99s	43.77s (+2%)	48.44s (+13%)	62.69s (+46%)	48.52s (+13%)
langford	31.10s	31.42s (+1%)	33.92s (+9%)	35.99s (+16%)	33.94s (+9%)

Table 1: Experimental results describing the overhead of the reification framework

low (below 3%). The computational cost can of course become much higher for expensive entailment/disentailment conditions, but it remains reasonable, especially if using watched literals: in our case below 13%.

Note that the overhead or benefits of propagating the negated constraint have not been really evaluated, but only the reification costs for checking entailment and disentailment. These experiments thus provide a preliminary evaluation mostly concerned with the overhead of our reification scheme. Nevertheless this restricted scope allows us to demonstrate a proof of concept. A deeper evaluation is planned using using our Rules2CP modeling language where all relations defined by rules are compiled to reified constraints, including global constraints [13].

5 Related Work

5.1 Half-Reification by Flattening

In [14], the authors remark that reified global constraints are not implemented because a reified constraint $B \leftrightarrow c$ must also implement a propagator for $\neg c$ (in the case that B = false). While for some global constraints, e.g. alldifferent, this may be reasonable to implement, for most the task seem very difficult, e.g. cumulative. They provide the following example (using Zinc syntax):
`constraint i <= 4 -> alldifferent([i,x-i,x]);`
The usual flattened form would be

```
1 constraint b1 <-> i <= 4;  % b1 holds iff i <= 4
2 constraint minus(x,i,t1);  % t1 = x - i
3 constraint b2 <-> alldifferent([i,t1,x]);
4 constraint b1 -> b2        % b1 implies b2
```

They remark that no solver currently implements the third primitive constraint.

Our scheme does allow the modeller to directly use full reification, since it provides the propagator for ¬`alldifferent` and the entailment checks needed for the third constraint.

Moreover, in the half-reified solution proposed in [14], one obtains:

```
constraint b1 -> i > 4;
constraint minus(x,i,t1);
constraint b2 -> alldifferent([i,t1,x]);
constraint b1 \/ b2
```

This encoding alleviates the need for a propagator for the negated global constraint. However, to fully benefit from propagation in the third constraint, one

needs to be able to test for the entailment of the negation of `alldifferent` (if `alldifferent` is false, then `i > 4` should be propagated). That is also answered by our framework.

5.2 Reification through Functional Dependencies

In [4], the authors remark that most global constraints can be reformulated as a conjunction of pure functional dependency (PFD) constraints with a constraint that can be easily reified. A PFD constraint is a constraint of the form $\mathbf{x}\ R\ f(\mathbf{y})$ where $\mathbf{x}$ and $\mathbf{y}$ are the free variables of the constraint, f is a function and R is a simple relation (e.g. equality) that can be reified.

This is in fact the scheme we have used in the previous section for the *cycle*, *global_cardinality* and *permutation* constraints.

It is worth noting however that the PFD-based reification, though it defines in a generic way the reification of a global constraint, does not fulfill three practical concerns:

- the level of consistency of the negated constraint obtained by decomposition can be much lower than what is possible to achieve;
- the computational cost of the negated constraint can also be significantly higher;
- reification is not just negating constraints but also detecting efficiently entailment and disentailment of global constraints.

These points can be illustrated on the *alldifferent* example. Its reification defined in [4] using its functional dependencies is a composition of the *sort* PFD constraint and the $<$ constraints:

$$(\exists y_i\ sort([x_i],[y_i]) \wedge y_1 < \cdots < y_n) \leftrightarrow B$$

The constraint to post if $B = 0$ is thus the conjunction of a *sort* and of the negation of the strict increasing constraint of the obtained variables, whereas *alldifferent* can also be, as we propose, directly negated as an *atmost_nvalue* global constraint. Moreover, the PFD-based reification leaves open the detection of entailment or disentailment of *alldifferent*. In Section 3.1 we have described a family of entailment conditions for that constraint and their efficient implementation using watched literals.

On the cumulative constraint, and on the family described as *logic* in [4], the PFD scheme leads to disjunctions of elementary constraints which cannot be efficiently propagated. On the other hand, our reification only involves the single global constraint *cumulatives* for which much better propagation can be achieved [3].

6 Conclusion

The reification of global constraints has been neglected up to now. It was believed that most global constraints would be hard to negate or test for entailment or disentailment but we have shown that this is not the case.

We have shown that for a large variety of global constraints, including the cumulative constraint, the negation of the constraint and the conditions of entailment can be efficiently expressed with a conjunction of other constraints or simple domain checks. Furthermore, we have given several examples where arc consistency is achieved for the negation of the constraint and where the entailment and disentailment conditions are domain-complete. Global constraints with extra variables, like for instance variables for end-time in addition to start time and duration, can also be negated efficiently with a simple case of constructive disjunction for eliminating the negation of the extra constraints.

However, sme difficulties remain. We have shown that the negation of the *diff-n*, i.e. the *overlap* constraint, does not admit any obvious expression without disjunctive constraints, and does not admit either any obvious efficient domain filtering algorithm. Although not coming directly from the modeling of natural combinatorial problems, we argue that the global constraint *overlap* is worth studying in its own right for the reification of *diff-n* and for the implementation of high-level modeling languages including *diff-n*.

Finally, the reification scheme that we have presented for global constraints can be applied to any constraint domain, not necessarily finite or discrete. In particular, it can be used for reifying hybrid discrete-continuous global constraints which now appear worth studying in many applicative contexts, e.g. for packing problems with complex shapes.

Acknowledgement.

We are grateful to Charles Prudhomme for his assistance on Choco, to Raphaël Martin for sharing his practical knowledge on global constraints and for his work on the implementation of Rules2CP for Choco, and to Thierry Martinez for interesting discussions on watched literals. This work is supported by the French ANR project Net-WMS-2.

References

[1] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraints catalog. Technical Report T2005-6, Swedish Institute of Computer Science, 2005.

[2] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraints catalog, 2nd edition. Technical Report T2012:03, Swedish Institute of Computer Science, February 2012.

[3] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Proceedings of the 8th Conference on Principles and Practice of Constraint Programming CP'02*, volume 2470 of *Lecture Notes in Computer Science*, pages 63–79. Springer-Verlag, September 2002.

[4] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. SICS Technical Report T2012:02, Swedish Institute of Computer Science, 2012.

[5] Nicolas Beldiceanu, Mats Carlsson, and Sven Thiel. Cost-filtering algorithms for the two sides of the sum of weights of distinct values constraint. SICS Technical Report T2002:14, Swedish Institute of Computer Science, 2002.

[6] Nicolas Beldiceanu, Irit Katriel, and Sven Thiel. Filtering algorithms for the same constraint. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 65–79. Springer-Verlag, 2004.

[7] Nicolas Beldiceanu, Xavier Lorca, and Thierry Petit. A GAC algorithm for a class of global counting constraints. Technical Report 10-01-INFO, École des Mines de Nantes, May 2010.

[8] Frédéric Benhamou, David A. McAllester, and Pascal Van Hentenryck. Clp(intervals) revisited. In Maurice Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming, SLP'94*, pages 124–138. MIT Press, 1994.

[9] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the NValue constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 884–888. Springer-Verlag, 2005.

[10] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - 9th International Conference, CP'03, Kinsale, Ireland, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer-Verlag, September 2003.

[11] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09*, pages 419–424, 2009.

[12] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* CP *'06)*, pages 700–705. Springer-Verlag, 2006.

[13] François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. In *Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP'08*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2009.

[14] Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half reification and flattening. In Jimmy Ho-Man Lee, editor, *Proceedings of CP 2011,*

17th International Conference on Principles and Practice of Constraint Programming, volume 6876 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2011.

[15] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[16] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming, CP'06*, pages 182–197, 2006.

[17] Ian P Gent, Ian Miguel, and Peter Nightingale. The alldifferent constraint: Exploiting exploiting strongly-connected components of other efficiency measures. Technical Report 2008/6, CIRCA - University of St Andrews, 2008.

[18] Lars Hellsten. Consistency propagation for stretch constraints. Master's thesis, University of Waterloo, 2004.

[19] Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in OPL. *ACM Transactions on Compututational Logic*, 1(2):285–320, 2000.

[20] Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of PADL'07*, pages 215–229. Springer-Verlag, 2007.

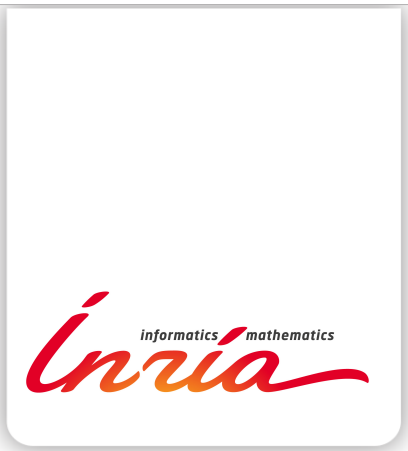
[21] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 1994*, pages 362–367, 1994.

[22] Jean-Charles Régin. Global constraints and filtering algorithms. In Michaela Milano, editor, *Constraints and Integer Programming, Toward a Unified Methodology*. Kluwer, 2004.

[23] Choco Team. CHOCO web page. `http//www.emn.fr/x-info/choco-solver/doku.php`.

[24] Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.

[25] Willem-Jan van Hoeve and Jean-Charles Régin. Open constraints in a closed world. In *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 3990 of *Lecture Notes in Computer Science*, pages 244–257. Springer-Verlag, 2006.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399