Efficient Modeling of Half-reified Global Constraints

Ignace Bleukx
KU Leuven, Department of Computer Science, Belgium

Hélène Verhaeghe
KU Leuven, Department of Computer Science, Belgium

Dimos Tsouros
KU Leuven, Department of Computer Science, Belgium

Tias Guns
KU Leuven, Department of Computer Science, Belgium

Abstract

Constraint modeling languages enjoy a wide applicability thanks to the ease of formulating complex relations between decision variables. One example is reified constraints, which enable reasoning over the activation of constraints, as done in Explainable Constraint Programming (XCP) algorithms, or the solving of arbitrarily nested expressions written by a user. However, in the case of global constraints, few solvers support them reified or even half-reified. To post the reification of a global constraint, we have to fall back to decomposing the constraint, thereby losing one of the key benefits of global constraints: the use of efficient propagators. In this work, we investigate how to reformulate any half-reified global constraint by means of auxiliary variables, while still using the stand-alone constraint and hence its efficient propagator. Our experiments show that this reformulation is competitive with solver-level support for half-reification, and is much more efficient compared to doing half-reification over a decomposition of the constraint.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases reification, global constraints, modeling, explainable constraint solving

Funding This research was partly funded by the Flemish Government (AI Research Program), the Research Foundation - Flanders (FWO) project G070521N and the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt)

1 Introduction

One of the features of Constraint Programming (CP) that sets it apart from other combinatorial solving paradigms is the notion of global constraints [42]. Global constraints allow a user to model high-level (n-ary) relations between variables which can capture the needed semantics for a variety of application domains. Moreover, the use of global constraints in a CP model provides the solver with high-level structural information about the problem. This allows for the use of efficient propagation routines tailored for these structures.

Arguably, the best-known global constraint is the AllDifferent constraint [41]. This constraint occurs in many types of problems: from Sudoku puzzles to real-world applications, like task allocation [20]. Many other global constraints have a more specific use. For example, the Cumulative constraint [1] enforces a set of tasks to be scheduled in such a way that the capacity of a machine is not exceeded; the Circuit constraint enforces an ordering of nodes in a graph such that they represent a Hamiltonian cycle [24]. CP-solvers exploit high-level structure present in global constraints using propagators: highly efficient algorithms that filter domains of variables while ensuring the constraints remain feasible during search.

When using constraint solvers through modeling languages, a typical workflow is as follows. The user writes a declarative constraint model in a high-level constraint modeling language. To find a solution to the constraints, the user selects a constraint solver and a *reformulation* system which compiles the high-level declarative model into a set of constraints posted to the solver, e.g., linear constraints for MIP solvers or unnested (global) constraints for CP solvers. Modeling languages and reformulation systems are often tightly integrated into a software package, for example MiniZinc [33], Conjure [2], PyCSP3 [26] or CPMpy [19].

Modeling languages allow users to write complex expressions, including reifications of global constraints. Reified constraints enable a user to link the truth value of a constraint to a Boolean variable that can then be used in another expression. However, solvers typically do not support the reification of global constraints, e.g., $b \rightarrow \textsc{AllDifferent}(x, y, z)$ is rarely supported by a solver [1]. Hence, such unsupported expressions should be rewritten to a set of constraints supported by the solver. For this, modeling systems typically decompose global constraints in order to post them to the constraint solver. E.g., the previous half-reification of the AllDifferent constraint is rewritten to $b \rightarrow (x \neq y \land x \neq z \land y \neq z)$, which is in turn rewritten to $b \rightarrow (x \neq y) \land b \rightarrow (x \neq z) \land b \rightarrow (y \neq z)$, as solvers typically do support reified (in)equalities. Decompositions of global constraints reformulate the relation into simpler expressions and are well-studied [7, 8, 13, 16], but a constraint solver will not use its dedicated propagation functions for the global constraint. Therefore, posting a decomposition is generally less efficient compared to posting a global constraint directly [9].

Besides a modeler explicitly writing (half-)reified global constraints, they can also occur as the result of *flattening* expressions [15]. Flattening is one of the core reformulations implemented in constraint modeling systems and converts any complex expression tree into a set of unnested constraints, as supported by the constraint solver.

► **Example 1** (Flattening [15]). Take the constraint $(i \leq 4) \lor \textsc{AllDifferent}(i, x - i, x)$. This constraint is unsupported by almost any constraint solver, and hence should be flattened into simpler constraints. The output of such a flattening algorithm can be the equivalent set of constraints: $\{b_1 \lor b_2, b_1 \rightarrow i \leq 4, b_2 \rightarrow \textsc{AllDifferent}(i, t_1, x), t_1 = x - i\}$.

Reified constraints are also commonly used in the field of eXplainable Constraint Programming (XCP) [11], where researchers investigate methods to help *explain* solutions or non-solutions of a constraint program to a user. For example, when no solution exists, explanation techniques can show a minimal conflicting set of constraints [10, 21, 23, 27, 28], or show the user a set of constraints which should be modified or relaxed [29, 30, 44]. Many algorithms used for generating such explanations rely on reasoning over subsets of constraints in the model [11, 32, 40].

Lastly, half-reification can be used as a way to solve Max-CSP problems. Similar to Max-SAT, the goal of Max-CSP solving is to find a solution satisfying a set of hard constraints while simultaneously maximizing the number of satisfied soft constraints. Related to this use-case is the notion of *soft global constraints* [43, 37]. Soft global constraints allow assignments which violate a the global constraint. E.g., the soft version of an AllDifferent constraint, may have some variables take the same value. In the case of Max-CSP solving, the goal is to minimize the magnitude of the violation (e.g., the number of variables taking the same value for AllDifferent). Half-reified constraints can be modeled using soft global constraints as well, by reifying the assignment of the violation to 0. However, soft-global constraints are uncommon in mainstream CP-solvers and have limited supported by most

[1] Two exceptions are the Choco solver [36] and the MINION solver [22]

modeling languages. Therefore, we restrict ourselves to the traditional "hard" sense of global constraints and reify those.

Motivated by the above use cases of reified constraints, the CP community has investigated several methods for reifying global constraints. These efforts include finding patterns for modeling full-reification of global constraints through decompositions [4], alternative ways to model the negation of global constraints for use in full-reification [14] and automatically deriving propagation routines for half-reified global constraints to use in solvers [15].

In this work, we build upon the findings of the above papers and contribute the following:

1. We prove the correctness of rewriting fully-reified total function constraints, as proposed but not proven in [4];
2. Inspired by the limited assumptions needed in the proof, we propose a generic and sound reformulation rule for modeling any *half*-reified global constraint using auxiliary variables;
3. We experimentally investigate the use of half-reified global constraints in two settings for generating explanations of unsatisfiable constraint programs, and we evaluate the efficiency of the (re)formulations across different solvers.

2 Background

A Constraint Satisfaction Problem (CSP) is a triplet ($\mathcal{V}$, $\mathcal{D}$, $\mathcal{C}$) with $\mathcal{V}$ a set of decision variables, $\mathcal{D}$ a set of integer domains associated with each variable and $\mathcal{C}$ a set of constraints. We write the domain of a variable as an enumerated set or an interval: $\mathcal{D}[x] = \{1, 2, 3\} = [1..3]$. If the domain of any of the decision variables is empty, it is called a *false domain*

A constraint $\mathrm{C} \in \mathcal{C}$ is a mapping of variable assignments to *true* or *false*. Variables occurring in a constraint are said to be in its *scope*. Constraints are written as a *predicate* with a list of arguments. E.g., the constraint $\textsc{AllDifferent}(x, y, z)$ uses the predicate $\textsc{AllDifferent}$ with the list of arguments $[x, y, z]$.

In addition to the set of decision variables $\mathcal{V}$, constraint models may contain *auxiliary variables* $\mathcal{A}$. These variables do not correspond to a user-introduced entity in the constraint problem and users generally do not care about their values [12, 38]. Auxiliary variables may be required to implement constraints more efficiently, e.g., to break symmetries [38], or to reformulate constraints in a way that the solver accepts them.

An assignment of variables *satisfies* a constraint if the constraint maps the assignment of the variables in its scope to *true*. A solution of a CSP is an assignment of each of the variables in $\mathcal{V}$ to a value in their domain, satisfying all constraints in $\mathcal{C}$. The set of all satisfying assignments to the decision variables $\mathcal{V}$ is written as $sols_{\mathcal{V}}(\mathcal{C})$. If a CSP does not allow a satisfying assignment (i.e., if $sols_{\mathcal{V}}(\mathcal{C}) = \emptyset$), it is *unsatisfiable*.

CP solvers use propagation functions, *propagators* in short, to *filter* values from the domains of variables based on the semantics of a constraint. A propagator $f_{\mathrm{C}}(\mathcal{D})$, for constraint C, is a monotonically decreasing function taking as input a set of domains $\mathcal{D}$ and returning a set of domains $\mathcal{D}'$: $\mathcal{D}'[x] \subseteq \mathcal{D}[x]$ for each variable $x \in scope(\mathrm{C})$. A propagation function $f_{\mathrm{C}}(\mathcal{D})$ is *domain consistent* if and only if for each variable pair $x, y \in scope(\mathrm{C})$, it holds that for any value in $\mathcal{D}'[x]$, there exists a value in $\mathcal{D}'[y]$ which satisfies the constraint [6]. E.g., given the $\textsc{Equals}$ constraint $a = b$ with $\mathcal{D}[a] = \{1, 2, 3\}$ and $\mathcal{D}[b] = \{1, 3, 4\}$, a *domain consistent* propagator will return domains $\mathcal{D}'[a] = \mathcal{D}'[b] = \{1, 3\}$. For simplicity, we assume the input of a propagation function is never a false domain.

A Boolean subexpression T in a constraint C is said to be in *positive context* if each solution of the constraint C is also a solution of the expression with T replaced by *true* [15]. The expression T is said to be in *negative context* if each solution of C is also a solution of

C with T replaced by *false*. Any expression not in positive nor negative context is said to be in *mixed context*.

▶ **Example 2** (Boolean contexts)**.** Below are several examples of constraints and types of contexts given with A and T any Boolean expressions and $\oplus$ the exclusive-or (XOR) operator:

Expression C	$\mathrm{A} \land \mathrm{T}$	$\mathrm{A} \lor \mathrm{T}$	$\neg\mathrm{T}$	$\mathrm{A} \rightarrow \mathrm{T}$	$\mathrm{A} \leftarrow \mathrm{T}$	$\mathrm{A} \leftrightarrow \mathrm{T}$	$\mathrm{A} \oplus \mathrm{T}$
Context of T	Positive	Positive	Negative	Positive	Negative	Mixed	Mixed

Global constraints

In CP, it is common to model complex relations between variables using *global constraints* [42]. Global constraints allow a user to model their problem in more natural ways compared to writing low-level constraints. Moreover, the use of global constraints provides additional information about the problem structure to the solver. This allows solver developers to write specialized *propagators* which work well with these specific structures [1, 25, 41]. The use of such propagators often leads to significant improvements in the runtime of the solver compared to decomposing the constraint. Decomposing a global constraint is the operation of rewriting the complex relation represented by the global constraint into semantically equivalent constraints, but "simpler" constraints. Modeling systems rely on the decomposition of global constraints whenever a (CP-) solver does not support a global constraint or whenever the global constraint occurs in an unsupported nesting.

▶ **Definition 3** (Functional constraint)**.** *A (global) constraint* $\mathrm{G}(\mathcal{V})$ *is a* functional constraint *if there exists a partitioning of the arguments* $\mathcal{V}$ *into two non-empty and non-overlapping subsets* $\mathcal{X}$ *and* $\mathcal{Y}$*, i.e.,* $\mathrm{G}(\mathcal{V}) = \mathrm{G}(\mathcal{X} \cup \mathcal{Y})$*, such that given any assignment to variables* $x \in \mathcal{X}$*, there exists at most one assignment to* $y \in \mathcal{Y}$ *such that the constraint is satisfied.*

A special case of functional constraints are *total functions*:

▶ **Definition 4** (Total function constraints)**.** *A* Total Function constraint (TF) *is a functional constraint for which there exists* exactly one assignment *for* $y \in \mathcal{Y}$ *for any assignment* $x \in \mathcal{X}$*.*

The global constraint catalog [5] names a functional constraint as a "pure functional dependency constraint" and contains 109 of such constraints[2]. An example of a total function constraint is $\textsc{Min}(v, X)$, commonly written as $\textsc{Min}(X) = v$, as for any values assigned to the variables in X, one can always compute the minimum and assign that unique value to v.

Reification

The reification of a constraint C relates the truth value of C to a Boolean variable b. Two common settings of reification are *full-reification* ($b \leftrightarrow \mathrm{C}$) and *half-reification* ($b \rightarrow \mathrm{C}$).

Full reification of a constraint is satisfied if the Boolean variable is asserted to *true if and only if* the constraint is satisfied. Hence, full-reification of a constraint requires modeling the negation of the constraint as well. For global constraints, modeling the negation can be non-trivial or inefficient, e.g., for the $\textsc{Circuit}$ constraint [4]. The authors of [4] identify common patterns to *decompose* fully-reified global constraints while [14] show the negation of some global constraints maps to other global constraints. Therefore, they argue the full reification of a global constraint should be split up into two *half-reifications*: $b \rightarrow \mathrm{C} \land \neg b \rightarrow \neg\mathrm{C}$.

[2] https://sofdem.github.io/gccat/gccat/Kpure_functional_dependency.html

The half-reification of a constraint $b \rightarrow \mathrm{C}$ is satisfied when $b = \mathit{true}$ and C is satisfied. If $b = \mathit{false}$, the value of C does not matter [45], i.e., $b \rightarrow \mathrm{C} \equiv \neg b \vee \mathrm{C}$. As this matches the notion of *material implication* from logic, we use *implication* and *half-reification* interchangeably in this paper. To refer to the Boolean variable b in a half-reification, we use the term *implication variable* or *indicator variable*. Half-reification does not require enforcing the negation of the constraint. Therefore, in order to construct a propagation function $f_{b \rightarrow \mathrm{G}(\mathcal{V})}$ for the half-reification of $\mathrm{G}(\mathcal{V})$ we can re-use the propagation function $f_{\mathrm{G}(\mathcal{V})}$ for G [15]:

$$\forall v \in \mathcal{V} : f_{b \rightarrow \mathrm{G}(\mathcal{V})}(\mathcal{D})[v] = \begin{cases} f_{\mathrm{G}(\mathcal{V})}(\mathcal{D})[v] & \text{if } \mathcal{D}[b] = \{\mathit{true}\} \\ \mathcal{D}[v] & \text{otherwise} \end{cases} \tag{1a}$$

$$f_{b \rightarrow \mathrm{G}(\mathcal{V})}(\mathcal{D})[b] = \begin{cases} \mathcal{D}[b] \setminus \{\mathit{true}\} & \text{if } f_{\mathrm{G}(\mathcal{V})}(\mathcal{D}) \text{ is a false domain} \\ \mathcal{D}[b] & \text{otherwise} \end{cases} \tag{1b}$$

Informally, if the implication variable is asserted to be *true*, the reified constraint should hold and its propagator is invoked (1a). If constraint C does not admit any solution given the domains, the implication variable should be asserted to *false* (1b). Note that the latter case can also be implemented using a *checker* instead of invoking the propagation function of the constraint. This is precisely what is proposed in [15], but rarely implemented in practice.

3 Full reification of global constraints

Modeling languages allow a user to construct any complex combination of constraints. This includes fully-reified global constraints with predicate G and arguments $\mathcal{V}$: $b \leftrightarrow \mathrm{G}(\mathcal{V})$. Apart from the Choco solver [36], and the MINION solver [18], no CP solver we are aware of systematically supports the reification of every global constraint it implements. Therefore, modeling reformulation systems rely on the decomposition of global constraints to model reified global constraints for all other solvers. In general, the decomposition of a global constraint may require auxiliary variables $\mathcal{A}$ to define the exact relation. The *definition* of how the auxiliary variables relate to the decision variables should always be satisfied, and hence cannot be a part of the reification. For any global constraint, a set of *defining* (F) and *conditioning* (K) constraints can be identified, with F a total function, uniquely defining the value for $\mathcal{A}$, given any value for $\mathcal{V}$ [3, 4]. As proposed by [4], this allows to write the full-reification of the global constraint to:

$$b \leftrightarrow \mathrm{G}(\mathcal{V}) \;\equiv\; \mathrm{F}(\mathcal{V} \cup \mathcal{A}) \wedge (b \leftrightarrow \mathrm{K}(\mathcal{V} \cup \mathcal{A})) \tag{2}$$

The solver is assumed to have support for the reification of K which can therefore only contain "core reifiable constraints" such as simple comparisons or clauses [4]. In practice, K is often the decomposition of a global constraint, as illustrated in the following example:

▶ **Example 5** (Full reification of Circuit)**.** Given the global constraint $\textsc{Circuit}(X)$ with X a list of three nodes: $X = [x_1, x_2, x_3]$, its full reification $b \leftrightarrow \textsc{Circuit}(x_1, x_2, x_3)$ can be written by introducing auxiliary "ordering" variables o_i, which indicate for each node when it is visited. We write: $(o_1 = x_1 \wedge o_2 = X[o_1] \wedge o_3 = X[o_2]) \wedge b \leftrightarrow \textsc{AllDifferent}(o_1, o_2, o_3)$. In this formulation, $\mathrm{F} = (o_1 = x_1 \wedge o_2 = X[o_1] \wedge o_3 = X[o_2])$ which *defines* the value of the ordering variables. These are then used in $\mathrm{K} = \textsc{AllDifferent}(o_1, o_2, o_3)$ to constrain the auxiliary variables. Note that here, the conditioning constraint is again a global constraint and will in practice also need to be decomposed. The constraint posted to the solver is: $o_1 = x_1 \wedge o_2 = X[o_1] \wedge o_3 = X[o_2] \wedge b \leftrightarrow (o_1 \neq o_2 \wedge o_1 \neq o_3 \wedge o_2 \neq o_3)$. Decomposing the AllDifferent constraint does not require introducing any auxiliary variables.

However, relying on the decomposition of a global constraint is not efficient from a solving perspective [9], and should be avoided, especially when many variables are involved. The authors of [3] notice that for total function constraints, i.e., $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$ as introduced in Definition 4, no decomposition is required to post the reification. Instead, the reification can be written by re-using the predicate G as the defining constraint and introducing auxiliary variables for the functionally defined variables $\mathcal{Y}$. The conditioning constraints K is then a set of channeling constraints linking the auxiliary variables $\mathcal{A}$ to the functionally dependent variables $\mathcal{Y}$. This is summarized by the following reformulation **AuxFullReifGlobal-TF**:

Reformulation of $b \leftrightarrow \mathrm{G}$ when G is a total function (**AuxFullReifGlobal-TF**)

$$b \leftrightarrow \mathrm{G}(\mathcal{V}) \equiv b \leftrightarrow \mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (b \leftrightarrow \mathcal{Y} = \mathcal{A}) \tag{3}$$

While not explicitly stated, it is clear the domains of $a \in \mathcal{A}$ should be chosen such that all solutions of the original constraint are kept, in order to satisfy the reified constraint when b is asserted to *true*. Additionally, the domains of $\mathcal{A}$ should also be chosen such that $\mathrm{G}(\mathcal{X} \cup \mathcal{A})$ admits at least one solution as this constraint is posted to the solver and hence always needs to be satisfied. We illustrate this approach in Example 6 considering the reification of the Min constraint.

▶ **Example 6** (Full reification of the Min constraint)**.** Consider the total function constraint $\textsc{Min}(v, X)$ which *defines* the value of variable v as the minimum of the variables in X. Then $\mathcal{X} = X$ and $\mathcal{Y} = \{v\}$. Following the rewrite rule above, we can rewrite $b \leftrightarrow \textsc{Min}(v, X)$ to $\textsc{Min}(v', X) \wedge (b \leftrightarrow v = v')$ with v' an auxiliary variable with domain $\mathcal{D}[v'] = \bigcup_{x \in X} \mathcal{D}[x]$.

We now present our first contribution and fully specify and prove **AuxFullReifGlobal-TF** (Eq. 3) as a sound reformulation for modeling the full reification of total function constraints, which is a fine-tuned version of the rules presented in [4].

▶ **Proposition 1** (Full reification of total function constraints)**.** *Given a total function constraint* $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$ *where all variables in* $\mathcal{Y}$ *are functionally dependent on* $\mathcal{X}$*, its reification* $b \leftrightarrow \mathrm{G}(\mathcal{X} \cup \mathcal{Y})$ *can be written as* $\mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (b \leftrightarrow \mathcal{Y} = \mathcal{A})$ *with* $\mathcal{A}$ *a set of auxiliary variables, allowing at least one solution for* $\mathrm{G}(\mathcal{X} \cup \mathcal{A})$ *and allowing all solutions of* $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$*.*

Proof. We distinguish two cases for the value of the Boolean variable b, and prove the reformulation is valid for both cases:

$$b = true: \quad true \leftrightarrow \mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (true \leftrightarrow \mathcal{Y} = \mathcal{A}) \tag{4}$$

$$\mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (\mathcal{Y} = \mathcal{A}) \tag{5}$$

As $\mathrm{G}(\mathcal{X} \cup \mathcal{A})$ allows all solutions of $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$, we can substitute $\mathcal{A}$ for $\mathcal{Y}$

$$b = false: \quad false \leftrightarrow \mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (false \leftrightarrow \mathcal{Y} = \mathcal{A}) \tag{6}$$

$$\neg\mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (\mathcal{Y} \neq \mathcal{A}) \tag{7}$$

We prove the equivalence in Equation (7) holds using contradiction by showing the negation of its first argument is impossible: $\mathrm{G}(\mathcal{X} \cup \mathcal{Y}) \equiv \mathrm{G}(\mathcal{X} \cup \mathcal{A}) \wedge (\mathcal{Y} \neq \mathcal{A})$. This implies that for an assignment for the variables in $\mathcal{X}$, there exist at least two assignments for the variables in $\mathcal{Y}$ satisfying the constraint G. As this contrasts with the hypothesis that $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$ is a total function, this negated equivalence cannot hold and Equation (7) must be true.

As both cases are now proven, Proposition 1 holds. ◀

Notice that in the above proof, we did not exploit the fact that G is a total function constraint when b is assigned to *true*. This observation is used in the following section to describe a reformulation for the half-reification of *any* global constraint.

4 Half-reification with auxiliary variables

We introduce a generic reformulation for half-reified global constraints using auxiliary variables. Compared to **AuxFullReifGlobal-TF** (3), the reformulation proposed in this section is not restricted to total function constraints $\mathrm{G}(\mathcal{X} \cup \mathcal{Y})$, but applies to *any* global constraint $\mathrm{G}(\mathcal{V})$. We introduce an auxiliary variable for each of the arguments of the global constraint and post the global constraint with auxiliary variables to the top-level of the constraint model. This results in the reformulation as given in Equation (8):

Reformulation of $b \rightarrow \mathrm{G}(\mathcal{V})$ for any G (**AuxHalfReifGlobal**)

$$b \rightarrow \mathrm{G}(\mathcal{V}) \equiv b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A}) \equiv \mathrm{G}(\mathcal{A}) \wedge (b \rightarrow \mathcal{V} = \mathcal{A}) \tag{8}$$

The logic behind this approach is the following: when the implication variable is *true*, the auxiliary variables are asserted to be equal to the original variables. Hence, if the domains of $\mathcal{A}$ allow all solutions of $\mathrm{G}(\mathcal{V})$, the original constraint holds. In any other case, the equivalence $\mathcal{V} = \mathcal{A}$ on the right-hand side does not need to hold, and the auxiliary variables in $\mathcal{A}$ are unconstrained. Still, as $\mathrm{G}(\mathcal{A})$ is posted to the solver and is therefore always enforced, we require the domains of $\mathcal{A}$ to be chosen such that $\mathrm{G}(\mathcal{A})$ satisfies at least one solution.

In practice, we can ensure both conditions are met by copying the domains of the original variables and checking that the original global constraint has at least one solution, or by adding values corresponding to a dummy solution to the domains of the auxiliary variables. Of course, if we are certain the original constraint does *not* admit any solution, we can simply omit the half-reification all together and post the constraint $\neg b$ to the solver instead.

We illustrate our reformulation using the AllDifferent constraint in Example 7.

▶ **Example 7** (Half-reification of AllDifferent)**.** To post the half-reification constraint $b \rightarrow \textsc{AllDifferent}(x, y, z)$ to a solver, we introduce new auxiliary variables x', y' and z' and rewrite the constraint to $\textsc{AllDifferent}(x', y', z') \wedge \big(b \rightarrow (x = x' \wedge y = y' \wedge z = z')\big)$ instead of half-reifying the decomposition. We ensure all original solutions are kept by copying the domains of the original variables. To ensure the top-level constraint can always be satisfied, we either check if it admits a solution using the original domains, or allow a dummy solution. In this case we can ensure $\{x' \mapsto 1, y' \mapsto 2, z' \mapsto 3\}$ is a dummy solution by using the domains: $\mathcal{D}[x'] = \mathcal{D}[x] \cup \{1\}$, $\mathcal{D}[y'] = \mathcal{D}[y] \cup \{2\}$, $\mathcal{D}[z'] = \mathcal{D}[z] \cup \{3\}$.

We want to remark that this idea is not entirely new, and several expert modelers have already employed this idea when modeling specific half-reified global constraints.[3] Paper [37] implements similar ideas directly into the solving process, where global constraints are made *soft* by introducing auxiliary variables and watching their domain changes.

In contrast, our reformulations allow to formulate half-reified global constraints at the modeling level. This enables a user to use half-reified global constraints in any setting, for any solver supporting the "hard" global constraint. Our reformulation is not implemented

[3] See for example: `https://github.com/google/or-tools/issues/973#issuecomment-744434446` or Tips 5.3 and 7.2 of the Gecode manual [39]

structurally into any existing modeling system, nor systematically evaluated. In the next section we evaluate the efficiency of this reformulation compared to decomposing the global constraint.

5 Experiments

We aim to answer the following experimental questions:

EQ1. How does rewriting half-reified global constraints using auxiliary variables compare to decomposing the global constraint in terms of runtime?

EQ2. To what extent are our reformulations competitive with solver-native support for half-reified global constraints?

We test our reformulations in two settings of eXplainable Constraint Programming where half-reified (global) constraints are essential for developing efficient algorithms [11]. The first is the problem of calculating Max-CSPs in order to find a minimal relaxation of the problem (Maximum Satisfiable Subset), and the second is the problem of extracting an unsatisfiable subset using assumption-based solving over half-reified constraints. This can then be used as the basis for finding a *Minimal* Unsatisfiable Subset [28].

Reification-based Max-CSP

The Max-CSP problem consists of finding a solution that maximizes the number of satisfied *soft* constraints ($\mathcal{C_S}$) while satisfying a set of *hard* constraints ($\mathcal{C_H}$), similarly to Max-SAT [17]. Given a $CSP(\mathcal{V}, \mathcal{D}, \mathcal{C_H} \cup \mathcal{C_S})$, we encode the Max-CSP as a constraint optimization problem with constraints $\mathcal{C_H} \cup \{b_C \rightarrow C \mid C \in \mathcal{C_S}\}$ and objective function $\mathit{maximize} \sum_{C \in \mathcal{C_S}} b_C$. This is a common approach that works with any existing CP solvers, in contrast to the use of dedicated soft-global constraint propagators [37]. In this experiment, we measure the time spent by the solver to find an optimal Max-CSP solution for a set of unsatisfiable CSPs.

Assumption-based solving

Lazy-clause generation (LCG) solvers [34] like OR-Tools CP-SAT and Chuffed hybridize CP's constraint propagation with SAT solving. This allows them to support solving with assumption variables, inherited from the underlying SAT solver. By considering the half-reification of each constraint in the problem, we can *activate* a set of constraints by *assuming* their indicator variables to be *true*. Many practical implementations of XCP techniques rely on this technique as it allows extracting a subset of infeasible constraints directly from the solver. Assumption-based solving can also be used for iteratively testing whether a subset of the constraints admits a solution, without restarting the solver from scratch [40, 31, 30, 11]. In this experiment, we focus on the first setting and add all indicator variables as assumptions. We then measure the time spent by OR-Tools to prove the unsatisfiability of the problem.

5.1 Experimental setup

We now discuss the benchmark instances used and then the methods to be compared.

Benchmarks

Our benchmarks consist of unsatisfiable CSPs with global constraints among the most widely used, namely AllDifferent, Cumulative, GlobalCardinality and Circuit.

Room-assignment. This benchmark considers the problem of assigning hotel rooms to customers who book a room for a given period. We generate instances with 50,75 and 100 requests on a horizon of 30 days with a mean duration of 3 days and a standard deviation of 2 days. For each number of requests, we generate 150 instances, for a benchmark set of 450 models. The CSP to solve this problem consists of AllDifferent constraints that prevent assigning overlapping requests to the same room and is made unsatisfiable by limiting the number of available rooms to 90% of the number of required rooms.

RCPSP. We consider 410 instances of the `j60` benchmark available on the PSPlib website and model each instance using Cumulative constraints. The CSP is made unsatisfiable by limiting the makespan to 90% of the best reported lower bound on the website of PSPlib.[4] Both the Cumulative constraints and precedence constraints are considered as *soft* constraints, and the limitation on the makespan is considered a *hard* constraint.

Multiple TSPs. In this last benchmark, we use an artificial problem in order to generate CSPs containing multiple Circuit constraints. The problem consists of finding n tours in n predefined clusters of a set of points. The goal is to minimize the sum of distances traveled in all clusters and the problem is made unsat by limiting the objective to 90% of the optimum. We generate the benchmark by sampling 35 points on a 100x100 grid and vary the number of clusters between 2 and 10, which are defined by a K-means algorithm. For each n we generate 50 instances, for a benchmark set of 450 instances.

Methods under investigation

We compare how rewriting half-reified global constraints performs to alternative strategies such as decomposing or direct solver support for the half-reified. As the goal is not to compare solvers directly to each other, we report the results of the following strategies solver by solver in Section 5.2.

- The decomposition(s) of the implication (**Decomp**). The decompositions used are those present in CPMpy, and their mathematical formulations are provided in Appendix B. For Cumulative, we show the results for two classical decompositions (time or task-based).
- Our method (**AuxHalfReifGlobal**) where all arguments in the constraint are replace with an auxiliary variable
- Native support (**Native**) for the half-reification of the global constraint.

Only the Choco solver has native support for the half-reification of each of the global constraints in our benchmarks. Choco implements half-reification similar to the propagation function described in Section 2, while using a specialized checker to check whether the reified constraint can still be satisfied using the current partial assignment.

We implemented our approaches on top of the CPMpy [19] constraint modeling library v0.9.17 in Python 3.9. For solvers we use OR-Tools [35] version 9.8, the Choco solver [36] version 4.10.14 and the Gecode solver [39] version 6.3.0 through MiniZinc v2.8.2. As all code is written in Python, each solver is accessed through its Python interface (using a custom branch of `pychoco` v.0.1.1 and the stable version of `minizinc-python` v0.9.0).

The experiments were run on Ubuntu 20.04 LTS on a single core of an Intel(R) Xeon(R) Silver 4214@2.2Ghz with 128GB of RAM. For each experiment, we employ a time-out of 2 hours. All code and benchmarks will be made available upon acceptance of the paper.

[4] `https://www.om-db.wi.tum.de/psplib` as accessed on 31/01/2024

5.2 Results and discussion

We report the results of solving the Max-CSP problem of the unsat problems in Fig. 1-3 for all solvers. Fig. 4 shows the runtimes for solving under assumptions with OR-Tools' CP-SAT solver. We now interpret these results and answer the experimental questions.

EQ1: Rewrite vs Decompose

We consider both the Max-CSP and the Assumption-based solving experiments to investigate how decomposing the global constraint compares to our reformulations. We first focus on the **AuxGlobal minimal** (red) and **Decomp** (blue) series in the figures. For all experiments, reformulating the global constraint is several orders of magnitude faster compared to decomposing the global constraint. For example, in Figure 1, OR-Tools is able to solve all instances within 0.1s while 40% of the instances reached the time-out of 2h when decomposed. Similarly, for the the CIRCUIT benchmark in Figure 3, Choco is able to solve ±95% of the instances within the given time-limit, whereas using the decomposition results into solving only half of the instances solves One exception on these results is the RCPSP benchmark when run on the Gecode solver. This is because Gecode does not support the CUMULATIVE constraint where the demand of a task or capacity of the resource is a variable, which is the case in our **AuxHalfReifGlobal**. Therefore, the solver internally uses the task decomposition of the constraint. Finally, when solving with assumptions (Figure 4), the reformulation is several orders of magnitude faster compared to the decompositions for all of the benchmarks.

EQ2: Rewrite vs Native

As only the Choco solver has support for half-reified global constraints, we focus on the middle column of Figures 1-3.

For the **Room-assignment** and **RCPSP** benchmarks, the native implementation of the half-reification is marginally faster for sub-second solve times. For the CIRCUIT constraint in the **Multiple TSPs** benchmark, our reformulation is comparable with **Native** for all instances. It seems for these global constraints, when solving a Max-CSP problem, the implementation of specialized propagators for half-reified global constraints only leverages marginal gains compared to our solver agnostic, model level reformulations.

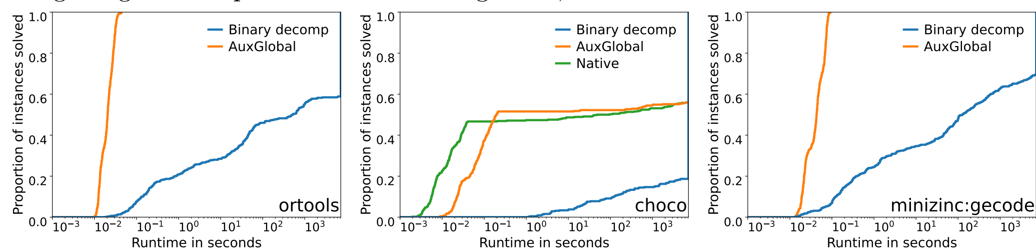


Figure 1 Runtimes of solving **Max-CSP** for the **Room-assignment** benchmark.

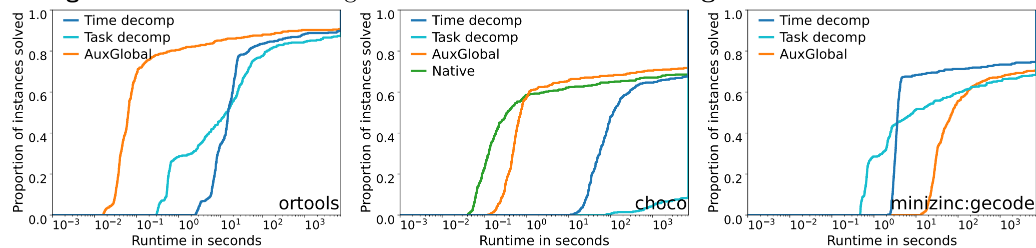


Figure 2 Runtimes of solving **Max-CSP** for the **RCPSP** benchmark.

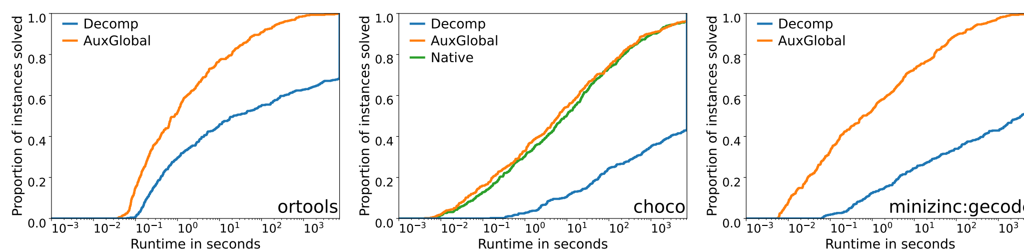
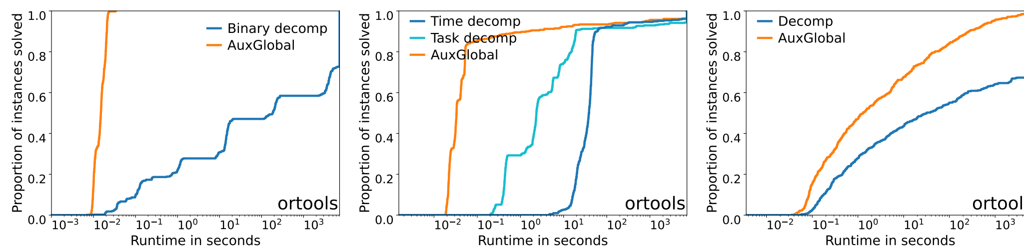


Figure 3 Runtimes of solving **Max-CSP** for the **Multiple TSP's** benchmark.



(a) Room Assignment **(b)** RCPSP **(c)** MultiTSP

Figure 4 Runtimes of **solving under assumptions** with OR-tools for all benchmarks

6 Conclusion

In this paper, we introduced and formalized several reformulations for half-reified global constraints. These reformulations allow to post a half-reification of a global constraint to any solver which supports the standard (non-reified) global constraint. Crucially, our reformulations avoid decomposing the global constraint, but instead introduce auxiliary variables and a reified channeling constraint. Therefore, it allows us to make use of the efficient propagators for global constraints found in CP solvers.

Our results show that the reformulations speed up the computation by several orders of magnitude compared to reifying a decomposition. Moreover, for several global constraints, the runtime for finding Max-CSP solutions to unsatisfiable constraint programs is similar to when the solver has native support for the half-reified global constraint.

In the future we want to take a closer look at *which* variables needs to be replaced for a sound reformulation of the reification. For example, in the case of total function constraints, we can follow the approach in [4] and replace only the functionally defined variables. Other questions arising include how to incorporate safening of partial functions and how to minimize the number of auxiliary variables introduced for non-functional constraints.

Finally, we want to compare our reformulations with more solver-native approaches such as those implemented in Max-CSP solvers with support for soft global constraints[5].

Overall, our work can contribute to a wider applicability of XCP techniques for CP problems, as it alleviates the runtime overhead of reifying decompositions, as well as having to restrict oneself to solvers that naively implement reified global constraints.

[5] e.g., toulbar2 `https://github.com/toulbar2/toulbar2`

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7):57–73, 1993.
2. Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.
3. Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. Technical report T2012:02, TASC team (INRIA/CNRS) and SCS and Uppsala University, Department of Information Technology, February 2012.
4. Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints An Int. J.*, 18(1):1–6, 2013. URL: `https://doi.org/10.1007/s10601-012-9132-0`, `doi:10.1007/S10601-012-9132-0`.
5. Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, (revision a), 2012.
6. Christian Bessiere. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006. `doi:10.1016/S1574-6526(06)80007-6`.
7. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. *arXiv preprint arXiv:0905.3755*, 2009.
8. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decomposition of the nvalue constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 114–128. Springer, 2010.
9. Christian Bessiere and Pascal Van Hentenryck. To be or not to be... a global constraint. In *International conference on principles and practice of constraint programming*, pages 789–794. Springer, 2003.
10. Ignace Bleukx, Jo Devriendt, Emilio Gamba, Bart Bogaerts, and Tias Guns. Simplifying step-wise explanation sequences. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: `https://doi.org/10.4230/LIPIcs.CP.2023.11`, `doi:10.4230/LIPICS.CP.2023.11`.
11. Ignace Bleukx, Tias Guns, and Dimos Tsouros. Explainable Constraint Solving: A hands-on tutorial, February 2024. `doi:10.5281/zenodo.10694140`.
12. Stéphane Bourdais, Philippe Galinier, and Gilles Pesant. Hibiscus: A constraint programming application to staff scheduling in health care. In *Principles and Practice of Constraint Programming–CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland, September 29–October 3, 2003. Proceedings 9*, pages 153–167. Springer, 2003.
13. Nicholas Downing, Thibaut Feydy, and Peter J Stuckey. Explaining alldifferent. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, pages 115–124, 2012.
14. François Fages and Sylvain Soliman. *Reifying global constraints.* PhD thesis, INRIA, 2012.
15. Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half reification and flattening. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2011. `doi:10.1007/978-3-642-23786-7_23`.
16. Thibaut Feydy, Peter J. Stuckey, and Mark Wallace. Why cumulative decomposition is not as bad as it sounds. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 746–761. Springer, 2009. `doi:10.1007/978-3-642-04244-7_58`.

17 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. `doi:10.1007/11814948_25`.

18 Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102. IOS Press, 2006. URL: `http://www.booksonline.iospress.nl/Content/View.aspx?piid=1654`.

19 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

20 Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, 2008.

21 Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 173–182. Springer, 2015. `doi:10.1007/978-3-319-23219-5_13`.

22 Christopher Jefferson, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429, 2010. URL: `https://doi.org/10.1016/j.artint.2010.07.001`, `doi:10.1016/J.ARTINT.2010.07.001`.

23 Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, volume 4. Citeseer, 2001.

24 Latife Genç Kaya and John N Hooker. A filter for the circuit constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 706–710. Springer, 2006.

25 Latife Genç Kaya and John N. Hooker. A filter for the circuit constraint. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 2006. `doi:10.1007/11889205_55`.

26 Christophe Lecoutre. PyCSP3, 04 2024. URL: `https://github.com/xcsp3team/pycsp3`.

27 Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016. `doi:10.1007/s10601-015-9183-0`.

28 João Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010*, pages 9–14. IEEE Computer Society, 2010. `doi:10.1109/ISMVL.2010.11`.

29 João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 615–622. IJCAI/AAAI, 2013. URL: `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6922`.

30 Carlos Mencía, Alexey Ignatiev, Alessandro Previti, and Joao Marques-Silva. Mcs extraction with sublinear oracle queries. In *Theory and Applications of Satisfiability Testing–SAT 2016:*

19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19, pages 342–360. Springer, 2016.

31 Carlos Mencía and Joao Marques-Silva. Efficient relaxations of over-constrained csps. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 725–732. IEEE, 2014.

32 Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 206–218. Springer, 2014.

33 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

34 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints An Int. J.*, 14(3):357–391, 2009. doi:10.1007/s10601-008-9064-x.

35 Laurent Perron and Vincent Furnon. Or-tools, 11 2022. URL: https://developers.google.com/optimization/.

36 Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, 6241:102–106, 2016.

37 Jean-Charles Régin. Using hard constraints for representing soft constraints. In Tobias Achterberg and J. Christopher Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, volume 6697 of *Lecture Notes in Computer Science*, pages 176–189. Springer, 2011. doi:10.1007/978-3-642-21311-3_17.

38 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

39 Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

40 Bart van Helvert. A solver agnostic incremental machine reasoning interface. Master's thesis, Eindhoven University of Technology, 2021.

41 Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. URL: https://arxiv.org/abs/cs/0105015.

42 Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

43 Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming (softening global constraints). *CoRR*, cs.AI/0408023, 2004. URL: http://arxiv.org/abs/cs.AI/0408023.

44 Stylianos Loukas Vasileiou, Alessandro Previti, and William Yeoh. On exploiting hitting sets for model reconciliation. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6514–6521. AAAI Press, 2021. URL: https://doi.org/10.1609/aaai.v35i7.16807, doi:10.1609/AAAI.V35I7.16807.

45 Mark Wallace and Mark Wallace. Constraint classes and solvers. *Building Decision Support Systems: using MiniZinc*, pages 85–111, 2020.

A Solver support for global constraints

In the table below, we summarize which solvers have support for the global constraints considered in this paper. A ✓ indicates the solver supports the global constraint at top-level of the constraint model. $\rightarrow$ means the solver has support for the half-reification of the global constraint.

Solver Interface	OR-Tools CPMpy	Choco CPMpy	Gecode MiniZinc
ALLDIFFERENT	✓	✓ $\rightarrow$	✓
CUMULATIVE	✓	✓ $\rightarrow$	✓
CIRCUIT	✓	✓ $\rightarrow$	✓
GLOBALCARDINALITY		✓ $\rightarrow$	✓

Table 1 Solver support for different global constraints.

B Decompositions of global constraints

AllDifferent

The binary decomposition of the half-reification $b \rightarrow \textsc{AllDifferent}(x_1, x_2, \ldots, x_n)$ is:

$$b \rightarrow \bigwedge_{i \neq j} x_i \neq x_j \tag{9}$$

Cumulative

Two well-known decompositions of the $\textsc{Cumulative}(S, D, E, H, c)$ constraint exist: the time-decomposition and task-decomposition. We model the half-reification of the time decomposition with n tasks as:

$$b \rightarrow \bigwedge_{t} c \geq \sum_{i=0}^{n} H_i \cdot (S_i \leq t \wedge E_i > t) \tag{10}$$

The task decomposition of the same constraint is defined as:

$$b \rightarrow \bigwedge_{j \in 1..n} c \geq \sum_{i=0}^{n} H_i \cdot (S_i \leq S_j \wedge E_i > S_j) \tag{11}$$

GlobalCardinality

We use the Boolean decomposition of the GLOBALCARDINALITY constraint inspired by [7]. In particular, the decomposition of $b \rightarrow \textsc{GlobalCardinality}(X, V, C)$ with $|X| = n$ and $|C| = |O| = m$ is defined as follows:

$$(\bigwedge_{x_i \in X} \sum_{j \in \mathcal{D}[x]} j \cdot a_{ij} = x_i) \wedge b \rightarrow \big((\bigwedge_{x_i \in X} \sum_{j \in \mathcal{D}[x]} a_{ij} \leq 1) \wedge (\bigwedge_{v_k \in V} \sum_{i=1}^{n} a_{iv_k} = O_k)\big) \tag{12}$$

with a_{ij} as set of auxiliary Boolean variables indicating whether variable x_i is assigned value j.

Circuit

Given the half-reification constraint $b \rightarrow \textsc{Circuit}(X)$ with $|X| = n$, we write the decomposition as follows:

$$o_1 = x_n \wedge \bigwedge_{i=2..n} o_i = X[o_{i-1}] \wedge b \rightarrow (o_n = 1 \wedge \bigwedge_{i \neq j} x_i \neq x_j \wedge o_i \neq o_j) \tag{13}$$

Where o_i are fresh “ordering” variables.