Efficient Reformulations of Half-reified Global Constraints using Auxiliary Variables

IGNACE BLEUKX, KU Leuven, Belgium
HÉLÈNE VERHAEGHE, UCLouvain, Belgium
DIMOS TSOUROS, KU Leuven, Belgium and University of Western Macedonia, Greece
TIAS GUNS, KU Leuven, Belgium

Background: In declarative constraint solving, a user formulates a constraint model in terms of variables and constraints, and uses a generic, off-the-shelf solver to solve the problem. Constraint Programming (CP) is one such constraint-solving paradigm, which has a wide support for many types of constraints, including *global constraints*. Global constraints capture complex relations between several decision variables, and CP solvers have specialized *propagators* to solve them efficiently.
Objectives: While CP solvers support a wide range of global constraints, few solvers support them in a *reified* or even *half-reified* context. Hence, for most CP solvers, reified global constraints must be decomposed into non-global constraints. This bypasses the global constraint propagator, which can considerably slow the solving process, suppressing one of the key benefits of modeling with global constraints: a faster solve-time. This is a problem for generic CP-modeling, as reified constraints are often introduced by the modeling system during translation and flattening of compound constraints. Additionally, half-reified global constraints are also used extensively in eXplainable Constraint Programming techniques (XCP). Therefore, XCP techniques suffer from scalability issues when global constraints occur in the model. In this paper, we aim to alleviate this bottleneck, allowing the use of the propagators of global constraints when modeling their half-reification, for all CP-solvers.
Methods: We propose a set of reformulation rules that allow the use of half-reification of a global constraint with any CP solver that supports the "normal" global constraint propagator. This is achieved by introducing auxiliary variables for the decision variables in the global constraint and the use of a reified channeling constraint, which most CP solvers support. Additionally, we show how to reduce the overhead of these auxiliary variables, by limiting the number of variables introduced, and by fixing their value when unconstrained. Finally, we prove the propagation strength of our reformulation for a variety of global constraint families.
Results: We experimentally evaluate the reformulations on a variety of global constraints and applications where reified constraints can occur. Our results show that this reformulation is much faster compared to decomposing the global constraint, and in some cases, even outperforms solver-level propagation routines for reified global constraints.
Conclusions: Using the reformulations proposed in this paper, we make the use of reified global constraints easily available for any CP solver. Hence, we expand the range of available solvers and constraint models that can be used in XCP techniques or for solving CSPs with compound constraints.

JAIR Associate Editor: Ken Brown

JAIR Reference Format:
Ignace Bleukx, Hélène Verhaeghe, Dimos Tsouros, and Tias Guns. 2026. Efficient Reformulations of Half-reified Global Constraints using Auxiliary Variables. *Journal of Artificial Intelligence Research* 86, Article 52 (August 2026), 33 pages. DOI: 10.1613/jair.1.21232

Authors' Contact Information: Ignace Bleukx, ORCID: 0000-0001-7810-8351, ignace.bleukx@kuleuven.be, KU Leuven, Leuven, Belgium; Hélène Verhaeghe, ORCID: 0000-0003-0233-4656, helene.verhaeghe@uclouvain.be, UCLouvain, Louvain-La-Neuve, Belgium; Dimos Tsouros, ORCID: 0000-0002-3040-0959, dtsouros@uowm.gr, KU Leuven, Leuven, Belgium and University of Western Macedonia, Kozani, , Greece; Tias Guns, ORCID: 0000-0002-2156-2155, tias.guns@kuleuven.be, KU Leuven, Leuven, Belgium.

DOI: 10.1613/jair.1.21232

1 Introduction

Constraint Programming (CP) (Rossi et al. 2006) is a declarative method for solving combinatorial problems, which is widely used to solve scheduling (Baptiste et al. 2006; Bleukx, Boumazouza, et al. 2025; Dejemeppe 2016; E. P. K. Tsang 2003a), (multi-dimensional) packing (Clautiaux et al. 2008; Iori et al. 2021; Pisinger and Sigurd 2007) and routing problems (Kaya and Hooker 2006a; Shaw 1998). One of the features of CP that separates it from other combinatorial solving paradigms, such as SAT (Biere et al. 2021), SMT or MIP-solving, is the notion of global constraints (Beldiceanu, Carlsson, and Rampon 2012; van Hoeve and Katriel 2006). Global constraints model high-level (n-ary) relations between variables, which can capture the needed semantics for various application domains. Modeling with global constraints allows users to represent complex relations between many decision variables using a single predicate. For example, the AllDifferent constraint can be used to enforce overlapping shifts to be assigned to different nurses in a rostering problem (E. Tsang et al. 2004), the Cumulative constraint (Baptiste et al. 2006) enforces a schedule of tasks without exceeding the machine's capacity and the Circuit enforces a Hamiltonian cycle in a graph (Kaya and Hooker 2006b).

CP solvers solve combinatorial problems using tree-based search and constraint *propagation* (Bessiere 2006). A constraint propagator filters the values of domains of decision variables based on the domains of other variables and the semantics of the constraint. As propagators are not limited to "simple" constraints such as clauses or linear inequalities, CP solvers can solve models that contain *global constraints* (van Hoeve and Katriel 2006). A CP model that uses global constraints provides the solver with high-level structural information about the problem. This allows to engineer efficient propagation mechanisms explicitly tailored for these structures. By using such efficient propagators during search, the number of alternative assignments can be reduced significantly, and many real-world problems can be solved efficiently when using global constraints. For example, efficient propagators for the Cumulative constraint have been shown to work extremely well for solving scheduling problems in industry (Schutt, Feydy, Stuckey, and M. G. Wallace 2011, 2010; E. P. K. Tsang 2003b).

In practice, CP solvers are often accessed through a modeling system. Such a system provides a high-level modeling language, e.g., Zinc (Nethercote et al. 2007), Essence (Frisch, Harvey, et al. 2008) or CPMpy (Guns 2019), which is agnostic to the underlying solver. This is useful from a user perspective, as it allows the modeler to focus on the modeling task rather than the solver's specifics. Typically, modeling languages allow a user to write more complex (combinations of) constraints compared to what the underlying solvers support. Hence, the modeling system is tasked with *transforming* the user-level model to a lower-level, solver-specific model before it can be solved (Akgün et al. 2022; Nethercote et al. 2007). For example, the model is *compiled* into a set of clauses when using a (Max)SAT solver, into a set of linear inequalities for MIP-solvers, or into a set of *flat* (global) constraints for CP solvers. This transformation phase of the constraint model offers many opportunities for the modeling system to enhance the model by rewriting or reformulating constraints. Examples of such enhancements include automatically breaking symmetries (Mears et al. 2015), eliminating common subexpressions (Nightingale et al. 2015; Rendl et al. 2008) or linearizing constraints (Belov et al. 2016), just to name a few. We consider two core transformations in modeling systems used for transforming constraint models for CP solvers: flattening of compound constraints and decomposing of unsupported global constraints.

Flattening compound constraints. When specifying a CP-model, a user may write any complex and nested expression in the modeling language. The system then has to split this complex expression tree into individual constraints supported by the solver. During this process, nested Boolean expressions are often rewritten into (half-)reified constraints (Feydy et al. 2011). The reification of a constraint links the truth-value of the constraint to a Boolean variable b. Two common settings are half-reification ($b \rightarrow \mathrm{C}$) and full-reification ($b \leftrightarrow \mathrm{C}$). Often, half-reification is enough to represent the user-level constraint at hand (Feydy et al. 2011). Consider the following example of a compound constraint consisting of two global constraints, which enforces that the values assigned to integer variables x, y, z have to be either different or all the same: $\textsc{AllDifferent}(x, y, z) \vee \textsc{AllEqual}(x, y, z)$. This

compound constraint is unsupported by most CP solvers and hence should be *transformed* into simpler constraints. An equivalent model is obtained by introducing two auxiliary variables (b_1 and b_2) and by rewriting the disjunction using two half-reified global constraints: $\{b_1 \rightarrow \textsc{AllDifferent}(x, y, z), b_2 \rightarrow \textsc{AllEqual}(x, y, z), b_1 \vee b_2\}$. Notably, most CP solvers do not support such (half-)reified global constraints, even if the "normal" version of the global constraint is implemented. For example, while most CP solvers implement the $\textsc{AllDifferent}(x, y, z)$ constraint, its half-reification $b_1 \rightarrow \textsc{AllDifferent}(x, y, z)$ is rarely supported (Feydy et al. 2011). Therefore, the modeling system must further simplify this expression using a decomposition of the reified global constraint.

Decomposing global constraints. For most global constraints, the exact relation it represents can be expressed using "simpler" elementary constraints. Modeling systems use such a *decomposition* whenever a global constraint is unsupported by the solver at hand, or when a global constraint occurs as part of an unsupported nesting. For example, consider again the constraint $b_1 \rightarrow \textsc{AllDifferent}(x, y, z)$. This reification constraint is typically unsupported by CP solvers, so the modeling system needs to rewrite it further into simpler constraints. Using the binary decomposition of the AllDifferent constraint, the modeling system rewrites the constraint to $b_1 \rightarrow (x \neq y \wedge x \neq z \wedge y \neq z)$ which is then further rewritten to $\{b_1 \rightarrow x \neq y, b_1 \rightarrow x \neq z, b_1 \rightarrow y \neq z\}$. This set of simpler expressions can be given to the solver, as most CP solvers typically do support the reification of comparison constraints, such as a disequality in this case. However, solving a model with decomposed global constraints is generally less efficient compared to using the global constraint propagators, as a global constraint allows reasoning over a more *global* structure, while the decomposition is often limited to reasoning over local relations in the model (Bessiere, Katsirelos, et al. 2009; Bessière and Van Hentenryck 2003). Therefore, using a decomposition of a global constraint should generally be avoided.

Applications of half-reified constraints. Apart from occurring as a result of flattening compound constraints, reified constraints are also commonly used in the field of eXplainable Constraint Programming (XCP) (Bleukx, Guns, et al. 2024), where researchers investigate methods to help *explain* solutions or non-solutions of a constraint problem to a user. For example, when no solution exists, explanation techniques can show a minimal conflicting set of constraints (Ignatiev et al. 2015; Junker 2001; Liffiton et al. 2016; Marques-Silva 2010), or show a set of constraints which should be relaxed (Marques-Silva, Heras, et al. 2013; Mencía et al. 2016; Vasileiou et al. 2021). Many algorithms for generating such explanations rely on reasoning over subsets of constraints, which can be implemented efficiently through incremental and assumption-based solving as implemented in modern Lazy-Clause Generation (LCG) solvers (Marques-Silva, Lynce, et al. 2021; Nadel, Ryvchin, and Strichman 2014b; van Helvert 2021; Wieringa 2014). For example, using the reification of each constraint in the model, one can repeatedly test the satisfiability of a subset of constraints by *assuming* the reification variables are set to *true* before solving, without restarting the solver from scratch. Similarly, to find a maximal subset of constraints that can be satisfied simultaneously, we can reify each constraint and maximize a (weighted) sum over the reification variables.

Contributions. 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 in decompositions for fully-reifying (functional) global constraints (Beldiceanu, Carlsson, Flener, et al. 2013), alternative ways to model the negation of global constraints for use in full-reification (Fages and Soliman 2012), and implementing propagation routines for (half-)reified global constraints directly in the solvers (Feydy et al. 2011; Jefferson et al. 2010; Lagerkvist and Schulte 2009).

In this paper, we investigate methods allowing to solve models with half-reified global constraints without decomposing, using any CP solver. More precisely, our reformulations allow to post the half-reification of any global constraint to any CP solver that implements the "normal" global constraint at the *top-level* of the constraint model.

We build on the findings of the above papers and contribute the following:

(1) We revisit the reformulation for fully-reified functional constraints, which was proposed but not proven by Beldiceanu, Carlsson, Flener, et al. (2013) and fully specify and prove its correctness.
(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 prove under which conditions this formulation is complete in terms of propagation strength and show how to minimize the number of required auxiliary variables; and
(4) We experimentally evaluate the use of our reformulations for flattening compound constraints with nested global constraints, and in two settings for generating explanations of unsatisfiable constraint models.

2 Related Work

We start by discussing approaches developed by the CP community for handling propagation of reified and half-reified global constraints in CP-solvers. To the best of our knowledge, only Choco (Prud'homme et al. 2016) and MINION (Gent, Jefferson, et al. 2006; Jefferson et al. 2010) implement a propagator for both the reification and half-reification of *all* of its (global) constraints. While Pumpkin (Flippo et al. 2024) supports half-reified constraints in their API, their propagators do not filter the indicator variable. That is, the only scenario where the half-reification propagates is when the Boolean indicator is *fixed* to *true*. The implementation of Choco and MINION is similar to the propagator described in Equation (1b), which essentially controls the execution of existing propagators based on the context of the search space. Similar ideas have been proposed and investigated by Lagerkvist and Schulte (2009) and Schulte (2000), but, to the best of our knowledge, they are not widely implemented in modern CP solvers. In theory, non-failing propagators (Björdal et al. 2020) can be used to implement half-reification, but we are not aware of any solver that does this.

Beldiceanu, Carlsson, Flener, et al. (2013) investigate common patterns among decompositions of fully-reified global constraints. Their methods for reifying *functional* global constraints are widely implemented in constraint modeling systems (Guns 2019; Nethercote et al. 2007) as a means of reformulating reified functional global constraints. Fully-reified constraints can be split into a half-reification of the constraint and a half-reification of its negation. Fages and Soliman (2012) observe that this negated global constraint can map to *other* global constraints. This allows for more efficient solving of fully-reified global constraints compared to decomposing them into non-global constraints. While their reformulations are implemented at the solver level, they can also be used at the modeling level if the underlying solver supports half-reified global constraints.

Half-reified constraints are often used when solving over-constrained problems, e.g., for finding a solution that satisfies a maximum number of constraints. The PFC-MRDAC algorithm (Larrosa et al. 1999) is a method for solving such Max-CSP problems. Régin (2011) adapts the PFC-MRDAC algorithm to work with global constraints. They replace the variables of the constraint with auxiliary ones, and during the execution of the PFC-MRDAC algorithm, the *Soft2Hard* method records when the domain of a decision variable becomes disjoint with the domain of an auxiliary one, which translates to a penalty on the resulting assignment. While the *Soft2Hard* system shares conceptual similarities to our approach due to a similar use of auxiliary variables, it is tailored specifically for the PFC-MRDAC algorithm. In contrast, our *modeling-level* solution is more generally applicable.

Lastly, and also related to solving over-constrained problems, is the notion of *soft global constraints* (van Hoeve, Pesant, et al. 2006). A soft global constraint can be violated to some extent, with violating assignments being valuated using a penalty function. E.g., the soft version of the AllDifferent constraint allows assigning variables to the same value, at some *cost*. Soft global constraints allow for a more fine-grained solution to over-constrained problems, compared to modeling with half-reified constraints, which simply turn the constraint on or off. Soft global constraints are available in cost-function-network solver Toulbar2 (De Givry 2023) and through the MiniBrass modeling system (Schiendorfer et al. 2018), but are not widely implemented in other CP solvers.

3 Background

A Constraint Satisfaction Problem (CSP) is a triplet $(\mathcal{V}, \mathcal{D}, C)$ with $\mathcal{V}$ a set of decision variables, $\mathcal{D}$ a set of discrete domains $\mathcal{D}[v]$ associated with each variable v, and C a set of constraints (Rossi et al. 2006). To solve a CSP, we aim to find an assignment to the variables $\mathcal{V}$, using values from their domains D, satisfying the set of constraints C. Depending on its domain, a variable is either Boolean ($\mathcal{D}[v] = \{true, false\}$) or integer ($\mathcal{D}[v] \subset \mathbb{Z}$). We represent the domain of a variable as an enumerated set or an interval. E.g., $\mathcal{D}[v] = \{1, 2, 3\}$ is equivalent to $\mathcal{D}[v] = [1..3]$. If for any variable $v \in \mathcal{V}$, $\mathcal{D}[v] = \emptyset$, then $\mathcal{D}$ is called an *empty domain*. We write the assignment of a variable v to a value x as $v \mapsto x$. We will often write $\mathcal{D}^{\mathcal{V}}$ as shorthand for $\mathcal{D}[v_1] \times \mathcal{D}[v_2] \times \cdots \times \mathcal{D}[v_n]$, i.e., $\mathcal{D}^{\mathcal{V}}$ is the cartesian product of all domains of the variables in $\mathcal{V}$, and represents the set of all assignments to $\mathcal{V}$.

A constraint C $\in C$ is an n-ary relation that maps variable assignments to *true* or *false*. Variables occurring in a constraint are said to be in its *scope*. We refer to the scope of C with *scope*(C). Constraints are often written as a *predicate* with an ordered set of arguments. For example, the constraint AllDifferent(x, y, z) uses the predicate AllDifferent with the arguments $\{x, y, z\}$. Assuming $\mathcal{D}[x] = \{1, 2\}$, $\mathcal{D}[y] = \{2, 3\}$, $\mathcal{D}[z] = \{1, 3\}$, the constraint represents the following relation:

$$\textsc{AllDifferent}(x, y, z) = \{(x \mapsto 1, y \mapsto 2, z \mapsto 3), (x \mapsto 2, y \mapsto 3, z \mapsto 1)\} \subset \mathcal{D}[x] \times \mathcal{D}[y] \times \mathcal{D}[z]$$

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 C. If a CSP does not allow a satisfying assignment, it is *unsatisfiable*. The set of all satisfying assignments to the decision variables $\mathcal{V}$ is written as $sols_{\mathcal{V}}(C)$. We write $sols(C)$ as shorthand for $sols_{scope(C)}(C)$. We write $C \equiv_{\mathcal{V}} C'$ when C and C' have the same set of solutions given a set of user-variables $\mathcal{V}$. If the set of user-variables is implicit from the context, we simply write $C \equiv C'$

Constraint modeling systems may introduce *auxiliary variables* before posting a constraint model to the solver. These variables do not correspond to a user-introduced entity in the constraint problem, and users generally do not care about their values (Bourdais et al. 2003; Rossi et al. 2006). Auxiliary variables may be required to implement constraints more efficiently, e.g., to break symmetries (Gent, Petrie, et al. 2006) or to reformulate constraints (e.g., (Belov et al. 2016)), as they are used in this work.

3.1 Boolean Subexpressions in Compound Constraints

In this paper, we will often consider *compound constraints*, i.e., constraints with multiple subexpressions. Depending on how a sub-expression occurs in the expression tree, it can occur in different Boolean *contexts* (Feydy et al. 2011). We consider three types of context: positive, negative, and mixed. 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* (Feydy et al. 2011). Hence, by replacing T in C by *true*, $sols(\text{C}) \subseteq sols(\text{C}_{true})$. Similarly, 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 a positive or negative context is said to be in *mixed context*. The context can be derived statically by traversing the expression tree (Feydy et al. 2011). Example 3.1 shows several examples of compound constraints and their context.

Example 3.1 (Boolean context of B). The table below shows the truth table for several compound constraints where A and B are both Boolean expressions. Based on the truth table, we can derive whether B occurs in a *positive*, *negative*, or *mixed* context. For example, for any satisfying solution α of A $\vee$ B, we can replace B $\mapsto$ *false* with B $\mapsto$ *true* to get a new satisfying assignment α'. Thus, B occurs in positive context in A $\vee$ B.

By exploiting the context of a subexpression, auxiliary variables can be linked to the expression they replace using half-reification instead of always using full-reification (Feydy et al. 2011). Similar ideas have been successfully applied in transforming compound Boolean formulas to CNF using the Plaisted-Greenbaum improvement of the

Table 1. Example Boolean contexts for subexpression B

A	B	A ∨ B	A → B	¬B	A ← B	A ↔ B	A ⊕ B
True	True	True	True	False	True	True	False
True	False	True	False	True	True	False	True
False	True	True	True	False	False	False	True
False	False	False	True	True	True	True	False
Context of B:		Positive	Positive	Negative	Negative	Mixed	Mixed

Tseitin transformation (Plaisted and Greenbaum 1986). This transformation takes into account whether each sub-formula occurs positively or negatively in the formula.

3.2 Propagation-driven Solving

CP solvers implement constraints using propagation functions, *propagators* in short, to *filter* values from the domains of variables based on the semantics of a constraint (Bessiere 2006). That is, during search, the solver will make a *branching* decision impacting a domain of a given variable (e.g., by fixing it to a given value, or by removing a range of values from its domain). This change is then *propagated* to the other variables in the constraint model using the propagation functions of all constraints in the CSP. A constraint propagator determines which values to *filter* from variable domains based on the semantics of the constraint.

Definition 3.2 (Propagator). A propagator $f_C(\mathcal{D})$, 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 $x \in scope(C)$.

Propagators can have different levels of consistency, based on which values are removed from the domain (Bessiere 2006). The strongest form of consistency is *domain consistency*, sometimes called *generalized-arc-consistency* or *hyper-arc-consistency*.

Definition 3.3 (Domain consistent propagation). Let $f_C(\mathcal{D})$ be a propagation function that returns a set of filtered domains D'. Then f_C is *domain consistent* if and only if for every variable $v \in scope(C)$ and for every value $x \in \mathcal{D}'[v]$, there exists an assignment to the other variables in $scope(C)$ with values in $\mathcal{D}'$ such that the tuple satisfies C when $v \mapsto x$.

Consider the 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, and because failure is typically handled by solver mechanisms outside of propagators, we will assume the input of a propagator is never an empty domain.

3.3 Global Constraints

In CP, it is common to model complex combinatorial relations between variables using *global constraints* (van Hoeve and Katriel 2006). The use of global constraints provides additional information about the problem structure to the solver, which allows solver developers to write specialized *propagators* that work well with these specific structures (Aggoun and Beldiceanu 1992; Kaya and Hooker 2006a; van Hoeve 2001). Using such propagators often leads to significant improvements in the runtime of the solver compared to solving a decomposition of the constraint.

Decomposing a global constraint is the operation of rewriting the complex relation represented by the global constraint into a semantically equivalent but "simpler" set of constraints. Modeling systems rely on the decomposition of global constraints whenever a (CP) solver does not support a global constraint (i.e., does not have an implementation of a propagation function for this constraint) or whenever the global constraint occurs as part of a compound constraint.

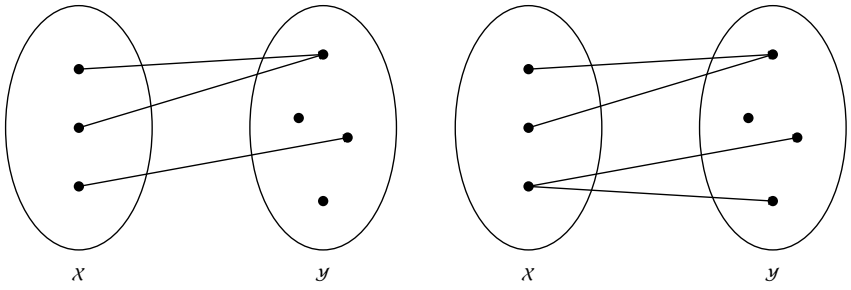

Fig. 1. Visualization of a total function constraint (left) and a total relation constraint (right)

As discussed above, a (global) constraint can be interpreted as a relation. Generalizing this, we can partition the arguments of a (global) constraint into two non-empty sets $\mathcal{X}$ and $\mathcal{Y}$. This re-interprets the constraint as a binary-relation where $\mathrm{G}(\mathcal{X}, \mathcal{Y}) \subseteq (\mathcal{D}[x_1] \times \mathcal{D}[x_2] \times \cdots \times \mathcal{D}[x_n]) \times (\mathcal{D}[y_1] \times \mathcal{D}[y_2] \times \cdots \times \mathcal{D}[y_m])$. This, in turn, allows us to reuse concepts from relational set theory, which we will use to specify the concepts of *total relation* and *total function* constraints.

Definition 3.4 (Total relation constraint). A constraint $R(\mathcal{X}, \mathcal{Y})$ is a *total relation* constraint if the following holds: $\forall X \in \mathcal{D}[x_1] \times \cdots \times \mathcal{D}[x_n] : \exists Y \in \mathcal{D}[y_1] \times \cdots \times \mathcal{D}[y_m] : R(X, Y)$

An example of a total relation constraint is $\textsc{AtMostNValue}(X, n)$ which is satisfied if the number of distinct values in X is at most n (Bessiere, Hebrard, et al. 2006). Indeed, for any value assigned to the variables in X, we can always determine a set of values to assign to n that satisfy the constraint. In particular, assigning n to any value in the interval $[\textsc{NValue}(X), \infty]$ satisfies the constraint.

Definition 3.5 (Total function constraint). A constraint $F(\mathcal{X}, \mathcal{Y})$ is a *total function* constraint if the following holds: $\forall X \in \mathcal{D}[x_1] \times \cdots \times \mathcal{D}[x_n] : \exists_{=1} Y \in \mathcal{D}[y_1] \times \cdots \times \mathcal{D}[y_m] : F(X, Y)$

That is, a total relation constraint is a total function, if for any assignment to variables $\mathcal{X}$, there exists **exactly one** assignment to $\mathcal{Y}$ such that the constraint is satisfied. An example of a total function constraint is $\textsc{Min}(X, y)$, commonly written as $\textsc{Min}(X) = y$, as for any value assigned to the variables in X, one can always compute the minimum value and assign that unique value to y.

The global constraint catalog (Beldiceanu, Carlsson, and Rampon 2012) names total function constraints as "pure functional dependency constraints" and contains 109 such constraints[1]. To the best of our knowledge, the concept of total relation constraints is undefined in related literature. Thus, we define it for the first time in this paper. We will use the notation $F(\mathcal{X}, \mathcal{Y})$ to indicate a total function constraint, and $R(\mathcal{X}, \mathcal{Y})$ for total relation constraints. When the exact type of global constraint is irrelevant, or the constraint is neither a total function nor a total relation, we simply write $\mathrm{G}(\mathcal{V})$.

3.4 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* ($b \rightarrow \mathrm{C}$).

The full-reification of a constraint $b \leftrightarrow \mathrm{C}$ links a Boolean variable b with a constraint C, such that b is true if and only if C is satisfied. Hence, full-reification of a constraint requires modeling the negation of the constraint as well, because the constraint should be asserted to be *false* when the Boolean variable is set to *false*. However,

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

in practice, it is often sufficient to consider half-reification of a constraint to express the relation at hand (Feydy et al. 2011).

The half-reification of a constraint $b \rightarrow \mathrm{C}$ links a Boolean variable b with a constraint C, such that if b is true, then C must be satisfied. If $b = \mathit{false}$, the value of C does not matter (M. Wallace and M. Wallace 2020) , and can even be undefined in the case of partial functions (Frisch and Stuckey 2009). Hence, the half-reification is always satisfied when b is assigned to *false*. We use the term *indicator variable* or *indicator* for short, to refer to the Boolean variable b in a half-reification.

A half-reification does not require enforcing the negation of the constraint for correctly implementing the propagator of the constraint. Therefore, as shown by Feydy et al. (2011), one can reuse the propagation function $f_{\mathrm{G}(\mathcal{V})}$ for implementing a propagator $f_{b\rightarrow \mathrm{G}(\mathcal{V})}$ for the half-reification of the constraint; following their notation:

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

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

Informally, if the indicator variable is set to be *true*, the reified constraint behaves as if it were asserted to be true (i.e., as if it was posted as a constraint at the top-level of the constraint model), and its propagator is invoked (1a). If there is no solution to the constraint C for the given variable domains, the indicator variable can be fixed to *false* (1b). Note that in practice, solver developers typically do not use the propagation function to check whether the constraint is falsified. Instead, they implement a specialized *checker* to check if the constraint is falsified under the given domains. Such a checker often has a lower runtime compared to the full propagation function, but is often weaker as well. To prove theoretical results in this paper, we will assume the propagation function of half-reified constraints is implemented exactly as above, with a (potentially domain consistent) propagator checking whether $f_{\mathrm{G}(\mathcal{V})}(\mathcal{D})$ is an empty domain. In Section 8, we will highlight some of the differences between these theoretical results and the empirically evaluated solver performances.

4 Full Reification of Global Constraints

Modeling languages allow to express any complex combination of constraints. This includes fully-reified global constraints with predicate G and arguments $\mathcal{V}$: $b \leftrightarrow \mathrm{G}(\mathcal{V})$. However, to the best of our knowledge, only Choco (Prud'homme et al. 2016), MINION (Gent, Jefferson, et al. 2006; Jefferson et al. 2010), and Pumpkin (Flippo et al. 2024) implement a generic reification mechanism to support the (half-)reification of *each* (global) constraint propagator implemented in the solver. Therefore, when using any other solver, modeling systems need to rely on decomposing global constraints to reformulate reified global constraints.

To formulate the decomposition of a global constraint, it may be necessary to introduce *auxiliary variables* $\mathcal{A}$. Beldiceanu, Carlsson, Flener, et al. (2013) observe that when the decision variables functionally define the auxiliary variables, this definition should be enforced at the top-level of the constraint model. Therefore, we write the decomposition of a global constraint using two sets of constraints: the *defining* (F) and *conditioning* (K) constraints. Here, $F(\mathcal{V}, \mathcal{A})$ is a total function, uniquely defining the values of the auxiliary variables $\mathcal{A}$, given any value for $\mathcal{V}$. Based on the value of the decision variables and the auxiliary variables, the reified constraint K controls the truth value of the global constraint. As proposed by Beldiceanu, Carlsson, Flener, et al. (2013), this allows writing the *full reification of any global constraint* to an expression of the following form:

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

In this setting, the authors assume the solver supports the reification of the constraint(s) in K. Therefore, K typically consists of simple constraints such as simple arithmetic constraints or clauses, which CP solvers support the reification of (Beldiceanu, Carlsson, Flener, et al. 2013). We illustrate the approach in the example below.

Example 4.1 (Full reification of ArgMin*).* The global constraint $\textsc{ArgMin}(X, i)$ assigns i to the index of a variable in X, which takes the minimal value among all variables in X. Note that ArgMin is not a functional constraint, as there may be multiple variables in X that take the minimum value. Its reification $b \leftrightarrow \textsc{ArgMin}(X, i)$ can be rewritten following the pattern in Equation (2) by introducing auxiliary variables v and w, resulting in the following formulation: $(\textsc{Min}(X, v) = v) \wedge (X[i] = w) \wedge (b \leftrightarrow (v = w))$. Here, the newly introduced variables v and w are *defined by* the functional[2] constraints $\textsc{Min}(X, v)$ and $X[i] = v$. Given these definitions of the auxiliary variables, the constraint $b \leftrightarrow (v = w)$ controls whether the ArgMin constraint is satisfied.

Beldiceanu, Carlsson, Flener, et al. (2012) identify several patterns on how the *defining* and *conditioning* constraints can be derived from the original global constraint. In particular, when the global constraint to be reified is a total function $F(\mathcal{X}, \mathcal{Y})$ itself, the predicate F can be re-used as the defining constraint by simply altering the arguments. The conditioning constraints K are then a set of channeling constraints linking the auxiliary variables $\mathcal{A}$ to the functionally dependent variables $\mathcal{Y}$. We illustrate this below using the functional constraint Min.

Example 4.2 (Full reification of the Min *constraint).* Consider the constraint $\textsc{Min}(X, v)$ which *defines* the value of variable v as the minimum of the variables in X. Here, $\mathcal{X} = X$ and $\mathcal{Y} = \{v\}$; and we can rewrite the reification $b \leftrightarrow \textsc{Min}(X, v)$ to $\textsc{Min}(X, v') \wedge (b \leftrightarrow (v = v'))$ with v' an auxiliary variable.

We summarize this special case of the reformulation in Equation (2) as **AuxFullReif-TF** (3). This reformulation is valid for any total function constraint $F(\mathcal{X}, \mathcal{Y})$ where variables $\mathcal{Y}$ are functionally defined by variables $\mathcal{X}$.

Reformulation of $b \leftrightarrow F(\mathcal{X}, \mathcal{Y})$ with F a total function (Beldiceanu, Carlsson, Flener, et al. 2012) **(AuxFullReif-TF)**

$$b \leftrightarrow F(\mathcal{X}, \mathcal{Y}) \equiv \underbrace{F(\mathcal{X}, \mathcal{A})}_{\text{Defining}} \wedge (b \leftrightarrow \underbrace{(\mathcal{Y} = \mathcal{A})}_{\text{Conditioning}}) \tag{3}$$

While not explicitly stated by Beldiceanu, Carlsson, Flener, et al. (2012), the above reformulation is only valid if the domains of $a \in \mathcal{A}$ are chosen such that all the solutions of the original constraint are kept, to satisfy the reified constraint when b is asserted to *true*. Additionally, the domains of $\mathcal{A}$ should also be chosen such that $F(\mathcal{X}, \mathcal{A})$ admits at least one solution for any assignment of $\mathcal{X}$. Indeed, in this reformulation, $F(\mathcal{X}, \mathcal{A})$ is always enforced and, as $\mathcal{X}$ may occur in other constraints in the model as well, the domains of the auxiliary variables $\mathcal{A}$ should allow a solution to G for any assignment of $\mathcal{X}$. As F is a total function, it is always possible to come up with such domains for $\mathcal{A}$.

We now present our first contribution by fully specifying and proving **AuxFullReif-TF** (3) as a sound reformulation for modeling the full reification of any total function constraint.

Proposition 4.3 (Full reification of total function constraints). *Given a total function constraint $F(\mathcal{X}, \mathcal{Y})$ where all variables in $\mathcal{Y}$ are functionally dependent on $\mathcal{X}$, its reification $b \leftrightarrow F(\mathcal{X}, \mathcal{Y})$ can be written as $F(\mathcal{X}, \mathcal{A}) \wedge (b \leftrightarrow (\mathcal{Y} = \mathcal{A}))$ with $\mathcal{A}$ a set of auxiliary variables with domains that allow all solutions of $F(\mathcal{X}, \mathcal{Y})$ and that allow at least one solution for $F(\mathcal{X}, \mathcal{A})$ for any assignment to $\mathcal{X}$.*

[2]Here we assume $1 \leq \mathcal{D}[i] \leq |X|$ so the Element constraint is a total function.

Proof. We prove the reformulation is valid for both values of b:

$$b \mapsto \mathit{true}: \quad \mathit{true} \leftrightarrow F(\mathcal{X}, \mathcal{Y}) \equiv F(\mathcal{X}, \mathcal{A}) \wedge (\mathit{true} \leftrightarrow (\mathcal{Y} = \mathcal{A})) \tag{4}$$

$$F(\mathcal{X}, \mathcal{Y}) \equiv F(\mathcal{X}, \mathcal{A}) \wedge (\mathcal{Y} = \mathcal{A}) \tag{5}$$

As $F(\mathcal{X}, \mathcal{A})$ is satisfiable, and allows all solutions of $F(\mathcal{X}, \mathcal{Y})$, we can substitute $\mathcal{A}$ for $\mathcal{Y}$

$$b \mapsto \mathit{false}: \quad \mathit{false} \leftrightarrow F(\mathcal{X}, \mathcal{Y}) \equiv F(\mathcal{X}, \mathcal{A}) \wedge (\mathit{false} \leftrightarrow (\mathcal{Y} = \mathcal{A})) \tag{6}$$

$$\neg F(\mathcal{X}, \mathcal{Y}) \equiv F(\mathcal{X}, \mathcal{A}) \wedge (\mathcal{Y} \neq \mathcal{A}) \tag{7}$$

We prove the equivalence in Equation (7) holds using contradiction by showing the negation of its first argument is impossible: $F(\mathcal{X}, \mathcal{Y}) \equiv F(\mathcal{X}, \mathcal{A}) \wedge (\mathcal{Y} \neq \mathcal{A})$. 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 F. As this contrasts with the hypothesis that $F(\mathcal{X}, \mathcal{Y})$ is a total function, this negated equivalence cannot hold, and instead, Equation (7) must be true. As both cases are now proven, Proposition 4.3 holds. □

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

5 Half-Reification With Auxiliary Variables

In this section, we introduce a generic reformulation for half-reified global constraints using auxiliary variables. Compared to **AuxFullReif-TF** (3), the reformulation proposed in this section is not restricted to total function constraints $F(\mathcal{X}, \mathcal{Y})$ but applies to *any* global constraint $\mathrm{G}(\mathcal{V})$. The reformulation works by introducing an auxiliary variable for each of the arguments of the global constraint and enforcing the global constraint with auxiliary variables at 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 (**AuxHalfReif**)

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

The logic behind this approach is the following: when the indicator 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 independent from $\mathcal{V}$. Still, as $\mathrm{G}(\mathcal{A})$ is 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 check whether the original constraint admits a solution and copy the domains of the original variables to the auxiliary variables. If the constraint does not admit a solution, we can simply enforce $\neg b$ without posting the reification at all. Such pre-processing is common in modeling systems such as MiniZinc, Conjure or CPMpy, where root-level propagation of a CP solver is used to simplify the constraint model before posting it to the solver. We illustrate our reformulation using the AllDifferent constraint in Example 5.1.

Example 5.1 (Half-reification of AllDifferent*).* Consider the half-reification $b \rightarrow \textsc{AllDifferent}(x, y, z)$. To reformulate this constraint, we introduce new auxiliary variables x', y', and z', and use the constraint $\textsc{AllDifferent}(x', y', z') \wedge \big(b \rightarrow (x = x' \wedge y = y' \wedge z = z')\big)$ instead of half-reifying the decomposition such as $b \rightarrow (x \neq y \wedge x \neq z \wedge y \neq z)$. All original solutions are kept by copying the domains of the original variables.

This reformulation allows to post the half-reification of any global constraint to a CP solver that supports the "top-level" global constraint, using its efficient propagator instead of decomposing it into smaller constraints. In the following proposition, we fully state our reformulation and prove its correctness.

Proposition 5.2 (Half-reification of global constraints). *The half-reification of any global constraint* $b \rightarrow \mathrm{G}(\mathcal{V})$ *can be written as* $b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A})$*, with* $\mathcal{A}$ *a set of auxiliary variables. By choosing the domains of these variables in such a way that* $\mathrm{G}(\mathcal{A})$ *is satisfiable and they allow all solutions of* $\mathrm{G}(\mathcal{V})$*, we can rewrite the constraint further to* $\mathrm{G}(\mathcal{A}) \wedge (b \rightarrow (\mathcal{V} = \mathcal{A}))$.

Proof. We first have to prove we can rewrite $b \rightarrow \mathrm{G}(\mathcal{V})$ to $b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A})$. We do this similarly to the proof of Theorem 4.3, using case-analysis on the Boolean variable b.

$$b \mapsto \mathit{true}: \quad b \rightarrow \mathrm{G}(\mathcal{V}) \equiv b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A})$$
$$\mathrm{G}(\mathcal{V}) \equiv \mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A}$$

$$b \mapsto \mathit{false}: \quad b \rightarrow \mathrm{G}(\mathcal{V}) \equiv b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A})$$
$$\mathit{true} \equiv \mathit{true}$$

Next, we need to prove that $b \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A})$ is equivalent to $\mathrm{G}(\mathcal{A}) \wedge (b \rightarrow (\mathcal{V} = \mathcal{A}))$

$$b \mapsto \mathit{true}: \quad \mathit{true} \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A}) \equiv \mathrm{G}(\mathcal{A}) \wedge (\mathit{true} \rightarrow (\mathcal{V} = \mathcal{A}))$$
$$\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A} \equiv \mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A}$$

$$b \mapsto \mathit{false}: \quad \mathit{false} \rightarrow (\mathrm{G}(\mathcal{A}) \wedge \mathcal{V} = \mathcal{A}) \equiv \mathrm{G}(\mathcal{A}) \wedge (\mathit{false} \rightarrow (\mathcal{V} = \mathcal{A}))$$
$$\mathit{true} \equiv \mathrm{G}(\mathcal{A})$$

Thus, if we chose the domains of the auxiliary variables such that $\mathrm{G}(\mathcal{A})$ is satisfiable, this last equation also holds and the reformulation is valid. □

We want to remark that this idea is not entirely new, and several expert modelers have employed this idea for modeling specific half-reified global constraints.[3] However, to our best knowledge, this idea has not yet been formalized or structurally implemented in any existing modeling system. Moreover, the performance of this reformulation is not systematically evaluated. In Section 6, we investigate this reformulation in detail and show how to minimize the overhead of the auxiliary variables. In Section 7 we discuss the propagation strength of this reformulation, and in Section 8, we evaluate the efficiency of this reformulation compared to decomposing the half-reification and to solver-level alternatives.

6 Reducing the Overhead of Auxiliary Variables

The previous section introduced a generic rewrite rule to model any half-reified global constraint without decomposing it. **AuxHalfReif** (Eq. 8) brings the global constraint to the top-level of the constraint model by replacing *each* argument of the constraint with an auxiliary variable, including those corresponding to a constant. This would introduce many auxiliary variables for high-arity global constraints. In general, adding variables to a constraint model is undesirable, as it may slow down the search due to more branching options or longer trails. We examine two methods for minimizing the overhead of the auxiliary variables: minimizing the total number of replaced variables and fixing the auxiliary variables during search when otherwise unconstrained.

[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 (Schulte, Tack, et al. 2010)

6.1 Minimizing the Number of Auxiliary Variables

According to the last case of the proof for Theorem 5.2, the reformulated global constraint with auxiliary variables should always be satisfiable. Our generic reformulation **AuxHalfReif** (8) complies with this requirement by replacing every decision variable with a fresh auxiliary one. However, in practice, it may not be required to replace all variables for the reformulation to be valid. Indeed, when the constraint at hand is a *total relation* $R(\mathcal{X}, \mathcal{Y})$ (see Definition 3.4), we can replace only a subset of the variables, namely $\mathcal{Y}$.

Minimal reformulation of $b \rightarrow R(\mathcal{X}, \mathcal{Y})$ for total relation constraint R (**AuxHalfReif-minimal**)

$$b \rightarrow \mathrm{G}(\mathcal{V}) \equiv b \rightarrow R(\mathcal{X}, \mathcal{Y}) \equiv R(\mathcal{X}, \mathcal{A}) \wedge (b \rightarrow (\mathcal{Y} = \mathcal{A})) \tag{9}$$

The intuition of this reformulation is similar to the case of full-reification of functional constraints. For any total relation constraint $R(\mathcal{X}, \mathcal{Y})$, we can find at least one satisfying assignment to variables $\mathcal{Y}$ given any assignment to $\mathcal{X}$. Hence, if we chose the domains of the auxiliary variables $\mathcal{A}$ in our reformulation such that all solutions of the original constraint are still valid, the reformulation is valid. Indeed, in that case we (1) allow all original solutions of the global constraints for when the indicator variable is set to *true* and (2) as the constraint is a total relation, the reformulation does not constrain the decision variables $\mathcal{X}$, i.e., the solver will always find a solution to $R(\mathcal{X}, \mathcal{Y})$ given *any* assignment to $\mathcal{X}$. Below, we provide several examples of total relation constraints that can benefit from such a minimal reformulation.

The $\textsc{Min}(X, y)$ constraint is a total function constraint which assigns the minimum of the values in X to variable y. As a total relation is a generalization of a total function, clearly **AuxHalfReif-minimal** holds for this constraint, and we can rewrite $b \rightarrow \textsc{Min}(X, y)$ to $\textsc{Min}(X, y') \wedge (b \rightarrow (y = y'))$ with y' an auxiliary variable.

The open version of the $\textsc{GlobalCardinality}(X, V, C)$ constraint is also a total function constraint, with multiple variables being *defined* at once, compared to only a single "output" variable in the Min from the previous example. X and C are sets of variables, and V is a set of values. The GlobalCardinality constraint is satisfied when the number of occurrences of value v_i in X is equal to the "counting" variable c_i. Hence, these counting variables C are functionally defined by the variables X and V by the function $c_i = \sum_{j=1..n} (x_j = v_i)$. Thus, for the reformulation, we can write $b \rightarrow \textsc{GlobalCardinality}(X, V, C)$ to $\textsc{GlobalCardinality}(X, V, C') \wedge (b \rightarrow (C = C'))$ where the C' is a set of freshly introduced auxiliary variables.

The $\textsc{ArgMin}(X, i)$ constraint is a total relational constraint as discussed previously. Indeed, given any values for the variables in array X, we can find at *least one* index on which the variable with the minimum value is located. Thus, to formulate the half-reification of this constraint, we can split up the variables to $\mathcal{X} = X$ and $\mathcal{Y} = \{i\}$. This leads to the following reformulation: $\textsc{ArgMin}(X, i') \wedge (b \rightarrow (i = i'))$ with i' an auxiliary variable.

The $\textsc{Cumulative}(S, D, H, c)$ constraint is often used in resource-constrained scheduling problems, and again represents a total relation. The arguments of the Cumulative constraint are sets of variables S, D, H which represent the start, duration, and resource demand of a set of tasks. The argument c is a variable that represents the capacity of the resource. The constraint is satisfied if the tasks are planned such that the concurrently running tasks never exceed the available resource capacity. We notice a total relation between the variables $\mathcal{X} = \{S, D, H\}$ and $\mathcal{Y} = \{c\}$. Indeed, given any values for the start, duration, and height variables, we can always find a capacity of the resource that allows that schedule of tasks. In particular, any value that allows all tasks to be run concurrently on the machine will satisfy the constraint. Therefore, we can write a half-reified Cumulative

constraint as follows: $\textsc{Cumulative}(S, D, H, c') \wedge (b \rightarrow (c = c'))$, if c' is an auxiliary variable for which the upper bound of c' allows for scheduling all tasks concurrently.

6.2 Forcing Auxiliary Variables to a Dummy Solution

The previous section describes strategies to minimize the number of introduced auxiliary variables when reformulating half-reified global constraints. The value of these auxiliary variables is determined by the semantics of the global constraint, the values of the original variables, and the domain of the indicator variable in the half-reification. Indeed, when the indicator variable is set to *true*, the global constraint acts as if it was enforced at the top-level of the constraint model. Hence, the domains of the auxiliary variables will mirror the domains of the original variables. Conversely, by setting the indicator variable to *false*, the auxiliary variables become independent from the ones they replaced, and are only influenced by the propagator of the global constraint $\mathrm{G}(\mathcal{A})$. Depending on the consistency level enforced by the propagator, this can lead to many branching decisions until a satisfying assignment is found, thereby slowing down the solving process.

However, as the value of the auxiliary variables is irrelevant from a user-perspective, we can alleviate some of the work done by the solver by fixing the auxiliary variables to a precomputed dummy solution α. This is shown in **AuxHalfReifDummy** where α is an assignment of variables $\mathcal{A}$ that satisfies G.

Reformulation with dummy solution of $b \rightarrow \mathrm{G}(\mathcal{V})$ for any G (**AuxHalfReifDummy**)

$$b \rightarrow \mathrm{G}(\mathcal{V}) \equiv \mathrm{G}(\mathcal{A}) \wedge (b \rightarrow (\mathcal{V} = \mathcal{A})) \equiv \mathrm{G}(\mathcal{A}) \wedge (b \rightarrow (\mathcal{V} = \mathcal{A})) \wedge (\neg b \rightarrow (\mathcal{A} = \alpha)) \tag{10}$$

AuxHalfReifDummy can be combined with **AuxHalfReif-minimal** (9). However, when replacing only a subset of the variables, we have to be careful that by introducing a dummy solution α for the auxiliary variables, we do not restrict the remaining original decision variables. For example, for functional constraints, introducing a dummy solution is not possible when using a minimal reformulation, as the remaining original decision variables uniquely define the value of the auxiliary variables.

In contrast, for total relation constraints, we can often combine both formulations. Consider the example of $\textsc{AtMostNValue}(X, n)$. Its half-reification $b \rightarrow \textsc{AtMostNValue}(X, n)$ is rewritten to $\textsc{AtMostNValue}(X, n') \wedge (b \rightarrow (n = n'))$. Here, we can safely introduce the assignment $\alpha = \{n' \mapsto |X|\}$ using constraint $\neg b \rightarrow (n' = |X|)$, without impacting the decision variables X. Indeed, $\textsc{AtMostNValue}(X, |X|)$ is always satisfied.

7 Propagation Strength

In the previous sections, we introduced a generic reformulation rule to model the half-reification of *any* global constraint, without decomposing it. The question remains, however, how well this reformulation works in practice. While Section 8 empirically answers this question, in this section we first investigate the propagation strength of our reformulation. The propagation strength of a (global) constraint largely determines how efficiently a CP solver can solve models containing such a constraint (Bessiere 2006; Bessière 1994; Schulte and Stuckey 2008). We now compare the propagation strength of our reformulation compared to when the solver has native support for half-reified global constraints.

Remember from Equation (1b) that the half-reification of a constraint can propagate in the following two cases:

(1) When the value of the indicator variable is fixed to *true* (Equation (1a)), then the constraint must be satisfied. This means that the reified constraint can now use its propagator directly, as if the global constraint were at the top-level of the constraint model.
(2) When the reified constraint does not admit a solution, given the current domains of the decision variables (i.e., when the propagator of the reified constraint *fails*), the value of the indicator variable is fixed to *false*.

We discuss each of these two cases in the following two subsections.

7.1 Propagation Strength when b is True

For our reformulation, the first case of the half-reification propagator is equivalent to the native half-reification propagator in Equation (1a). Indeed, given a domain consistent propagator for the Equals constraint (i.e., $\mathcal{V} = \mathcal{A}$), the domains of the auxiliary variables will be equal to the domains of the variables they replace. Formally,

$$f_{\mathrm{G}(\mathcal{V})}(\mathcal{D}) = f_{\mathrm{G}(\mathcal{A})}(f_{\mathcal{V}=\mathcal{A}}(D))$$

Thus, if the solver propagates until fix-point in each search node, the reformulated reified global constraint will behave as if it were posted at the top-level of the constraint model whenever in the search tree the indicator variable is fixed to *true*. This is identical to the first case of the reified propagator outlined in Equation (1a).

7.2 Filtering b to False

During search, the domains of the decision variables may be filtered such that the reified global constraint does not admit a solution. However, in our reformulation, we use an Equals constraint in the reification, which always admits a solution, except if the domains of the two variables it compares are disjoint. Therefore, our reformulation is not equivalent to the native propagator of half-reified global constraints from a propagation point of view. Indeed, while the propagator of the global constraint may detect there is no solution to the constraint given the current domains, it is not always possible to detect this in our reformulation through the Equals constraint. We illustrate this below with the AllDifferent constraint.

Example 7.1 (Propagation of reformulated AllDifferent*).* Consider the constraint $b \rightarrow \textsc{AllDifferent}(x, y, z)$ with domains $\mathcal{D}[x] = \mathcal{D}[y] = \mathcal{D}[z] = \{1, 2, 3\}$. In a particular branch of the search tree, we encounter the following domains: $\mathcal{D}[x] = \mathcal{D}[y] = \mathcal{D}[z] = \{1, 2\}$. Given these domains, $\textsc{AllDifferent}(x, y, z)$ cannot be satisfied, which will be detected by any domain consistent (or even bound consistent) propagator for AllDifferent. Therefore, after invoking this propagator of AllDifferent, the half-reification propagator will filter the Boolean indicator variable b to domain $\{\mathit{false}\}$ following Equation (1b). Now we compare this to our reformulation. Assume we copied the domains for our reformulation $\textsc{AllDifferent}(x', y', z') \land b \rightarrow (x = x' \land y = y' \land z = z')$ and thus the domains of the auxiliary variables are still $\mathcal{D}[x'] = \mathcal{D}[y'] = \mathcal{D}[z'] = \{1, 2, 3\}$ in that same branch of the search tree. In our reformulation, the reified Equals constraint $(x = x' \land y = y' \land z = z')$ **can** be satisfied given these domains. Thus, the indicator variable will not be fixed to *false* when using the reformulation.

The example shows that in general, our reformulation has a weaker level of consistency compared to the native implementation following Equation (1b). More precisely, whenever the following lemma is applicable, the native implementation will filter the Boolean variable to $\{\mathit{false}\}$ while our reformulation will not.

Lemma 7.2. *If there exists a set of domains $\mathcal{D}$ such that: (1) the propagator of* $\mathrm{G}(\mathcal{V})$ ***fails*** *(i.e., $f_{\mathrm{G}(\mathcal{V})}(\mathcal{D}) = \emptyset$), (2) the propagator for* $\mathrm{G}(\mathcal{A})$ ***does not fail*** *(i.e., $f_{\mathrm{G}(\mathcal{A})}(\mathcal{D}) = \mathcal{D}' \neq \emptyset$) and (3) the propagator for $\mathcal{V} = \mathcal{A}$* ***does not fail*** *for the newly derived domains (i.e., $f_{\mathcal{V}=\mathcal{A}}(\mathcal{D}') \neq \emptyset$), then* ***AuxHalfReif*** *has a weaker level of consistency compared to the propagator in Equation* (1b).

Proof. We prove this lemma by construction by filling in the domains in the half-reification propagator from Equation (8) and in our reformulation.

- For the half-reified propagator: from condition (1) of the lemma, we know that the propagator of the global constraint detects the constraint is unsatisfiable (i.e., $f_{\mathrm{G}(\mathcal{V})}(\mathcal{D}) = \emptyset$). Thus, *true* is removed from the domain of the Boolean indicator variable based on Equation 1b.

- For our reformulation: from conditions (2) and (3), we know that the reified channeling constraint $\mathcal{V} = \mathcal{A}$ is satisfiable even after G($\mathcal{A}$) propagates. Hence, *true* cannot be filtered from the domain of the indicator variable, and the solver cannot derive anything in the current search node. □

The proposition below determines sufficient conditions on the propagators used in our reformulation, for which the reformulation is equally strong compared to the native reified propagator.

Proposition 7.3. *Assume we have a total relation constraint with a domain consistent propagator $R(\mathcal{X}, \mathcal{Y})$, and a domain consistent propagator for the* Equals *constraint. If $\mathcal{Y} = \{y\}$, i.e., $\mathcal{Y}$ is a singleton set, then for any set of domains $\mathcal{D}$, it holds that*

$$(f_{R(\mathcal{X},\mathcal{Y})}(\mathcal{D}) = \emptyset) \wedge (f_{R(\mathcal{X},\mathcal{A})}(\mathcal{D}) = \mathcal{D}') \wedge (\mathcal{D}' \neq \emptyset) \Rightarrow f_{\mathcal{Y}=\mathcal{A}}(\mathcal{D}') = \emptyset$$

I.e., the conditions in Theorem 7.2 cannot be satisfied simultaneously and hence the reformulation has the same propagation strength compared to the native implementation.

Proof. Let's assume that the propagation function $f_{R(\mathcal{X},\mathcal{Y})}(\mathcal{D}) = \emptyset$. Given that R is a total relation constraint, we can always construct domains for variables in $\mathcal{A}$, such that $R(\mathcal{X}, \mathcal{A})$ admits a solution. Thus, after propagating $f_{R(\mathcal{X},\mathcal{A})}(\mathcal{D})$ we obtain a new set of domains $\mathcal{D}'$. Now, in the case where $|\mathcal{Y}| = 1$, this means that $\mathcal{D}^{\mathcal{Y}} = \mathcal{D}[y]$ and $\mathcal{D}^{\mathcal{A}} = \mathcal{D}[a]$. Additionally, it holds that for this new domain $\mathcal{D}'[a] \subseteq \mathcal{D}[a] \setminus \mathcal{D}[y]$. Indeed, if some $v \in \mathcal{D}'[a] \cap \mathcal{D}[y]$ remained after propagating $R(\mathcal{X}, \mathcal{A})$, then by domain consistency it has a support x in $\mathcal{D}[X]$, and the same tuple $(x, y = a)$ would be a support for $R(\mathcal{X}, \mathcal{Y})$, contradicting $f_{R(\mathcal{X},\mathcal{Y})}(\mathcal{D}) = \emptyset$. Therefore, it follows that the domain of a after propagation are disjoint with the domains of y and thus $f_{\mathcal{Y}=\mathcal{A}}(\mathcal{D}') = \emptyset$. □

We now discuss the two sufficient conditions stated in the lemma in more detail.

Requirement of domain-consistency for R. A domain consistent propagator for R is required to guarantee the proposition holds. The following example demonstrates how, for example, bound consistency may be insufficient. Consider again the constraint ArgMin($[x_1, x_2, x_3, x_4], i$), which is satisfied if i is assigned an index for which x_i is equal to the minimum value among the x variables. Using our minimal reformulation rule, the reification $b \rightarrow$ ArgMin($[x_1, x_2, x_3, x_4], i$) is rewritten to ArgMin($[x_1, x_2, x_3, x_4], i'$) $\wedge\ b \rightarrow (i = i')$. Assume we have a bound consistent propagator for ArgMin (Bessiere 2006), and the following set of domains $x_1 \in [0..4]$, $x_2 \in [5..6]$, $x_3 \in [5..6]$, $x_4 \in [0..4]$, $i \in [2..3]$ and $i' \in [1..4]$. For the original constraint, the propagator for ArgMin($[x_1, x_2, x_3, x_4], i$) will determine that there is no solution possible. As a result, the Boolean indicator variable is fixed to *false*. In our reformulation, no propagation on ArgMin is possible with a bound consistent propagator. Indeed, there is a solution for both the lower and upper bounds of i'. Additionally, $i = i'$ is satisfiable as their domains have a non-empty overlap; hence the native implementation of the half-reified constraint propagates more strongly compared to our reformulation.

Requirement of $|\mathcal{Y}| = 1$. When a total relation has more than one "output" variable, then there is no guarantee the proposition holds. As an example, consider the open version of GlobalCardinality($[x_1, x_2], [1, 2, 3], [c_1, c_2, c_3]$). This is a total function constraint, with multiple functionally defined variables c_i. Our reformulation of the half-reified version of this constraint is GlobalCardinality($[x_1, x_2], [1, 2, 3], [a_1, a_2, a_3]$)$\wedge b \rightarrow (c_1 = a_1) \wedge (c_2 = a_2) \wedge (c_3 = a_3)$. Assume the domains of the variables are: $\mathcal{D}[x_1] = \mathcal{D}[x_2] = [1..10]$, $\mathcal{D}[c_1] = \mathcal{D}[c_2] = \mathcal{D}[c_3] = [1..2]$ and $\mathcal{D}[a_1] = \mathcal{D}[a_2] = \mathcal{D}[a_3] = [0..2]$. A propagator for GlobalCardinality can detect that the constraint cannot be satisfied. Indeed, the sum of the counting variables can be at most 2, which is impossible given the domains. Hence, the propagator for the half-reification will filter b to *false*. In our reformulation, in contrast, $(c_1 = a_1) \wedge (c_2 = a_2) \wedge (c_3 = a_3)$ is satisfiable, and no filtering will occur.

Note that while the two conditions in the lemma are sufficient for our reformulation to have the same propagation strength as the native implementation, they are not necessary conditions. The precise propagation strength of the reformulation depends on the exact implementation of the global constraint's propagator. For example, if a propagator for GlobalCardinality does not check the relation between the counting variables, the negative example from above no longer holds, and the reformulation *does* have the same propagation strength as the native half-reified propagator from Equation (1b). Indeed, in the experiments in Section 8, the reformulation for GlobalCardinality achieves the same propagation strength as the custom half-reified propagator.

7.3 A Note on Checkers in Solvers

As proposed by Feydy et al. (2011) and shown in Equation (8), the propagator for the half-reification (global) constraints can be implemented using the standalone propagator of the constraint to be reified. However, in practice, solver developers use a constraint *checker* instead of using the propagator for checking if the constraint is satisfiable given the current domains. This is done either because of implementation details of the solver or because a *checker* algorithm is often faster compared to running the full propagator. However, for many global constraints, it is difficult to implement an efficient algorithm that checks whether the given domains allow a solution. Therefore, when the solver does not use the propagator to check if the constraint is satisfiable, it is common to *only* check for this if all variables in the constraint's scope are instantiated (Fages and Soliman 2012). When this is the case, our model-level reformulation can actually be *stronger* compared to such a solver-level implementation that waits for all variables to be instantiated. Naturally, our reformulation still requires that the solver implements a strong checker for the Equals constraint. Such a checker for the Equals constraint is easy to implement, as it only needs to check whether the domains of two variables are disjoint. Hence, it is implemented in most CP-solvers. In the following example, we provide an example where our reformulation can propagate stronger compared to what is implemented in solvers in practice.

Example 7.4 (Constraint check of AllDifferent*).* Consider again the half-reification $b \rightarrow \textsc{AllDifferent}(x, y, z)$ where the original domains of the decision variables are $\mathcal{D}[x] = \mathcal{D}[y] = \{1, 2\}$ and $\mathcal{D}[z] = \{1, 2, 3, 4\}$. This constraint will be rewritten to $\textsc{AllDifferent}(x', y', z') \wedge b \rightarrow (x = x' \wedge y = y' \wedge z = z')$ where the auxiliary variables have the same domains as the decision variables they replace. After propagating the AllDifferent constraint the domain of z' is filtered to $f_{\textsc{AllDifferent}(x', y', z')}(\mathcal{D})[z'] = \{3, 4\}$.

Assume that during search, the domain of z is filtered to $\{1, 2\}$ due to other constraints in the CSP. Now, the AllDifferent constraint cannot be satisfied, and hence the indicator variable should be fixed to *false*. However, because none of the decision variables are assigned to a value, a constraint checker as described above will not report this. In contrast, we can easily check if the constraint $z = z'$ is satisfiable or not, by computing the intersection of both domains, and hence, in our reformulation, b will be filtered to $\{\mathit{false}\}$ in this setting.

8 Experimental Evaluation

We evaluate these reformulations of half-reified global constraints in different application domains and benchmarks to answer the experimental questions listed below. For the first four, we focus on efficiency when solving with half-reified global constraints, focusing on computing a Max-CSP solution or solving a problem with compound constraints where global constraints occur as an argument to another expression.

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

EQ2 How does a reformulation with a reduced number of auxiliary variables compare to replacing all variables?

EQ3 How does forcing the auxiliary variables to a dummy solution when the indicator variable is fixed to *false* impact the runtime of the solver?

EQ4 How do our reformulations compare with solver-native support for half-reified global constraints?

EQ5 How does our reformulation impact the extraction of UNSAT cores from the solver in terms of runtime and core-size?

Due to the limited support of reified global constraints by solvers, competitions such as the MiniZinc challenge (Stuckey et al. 2014) and XCSP3 competition (Audemard, Lecoutre, et al. 2024) rarely include instances with them. Therefore, we introduce the following application domains used to answer the above experimental questions.

Application 1: Compound constraints in positive context. The first setting of half-reified (global) constraints that we consider is that of flattening complex expression trees. As discussed by Feydy et al. (2011), any Boolean expression that occurs as part of a compound constraint, but in a positive context, can be *flattened* using half-reification. That is, the subexpression can be replaced with a Boolean variable b, and the half-reification of the expression can be added to the top-level of the constraint model. In this experiment, we measure the time the solver spends to find a satisfying solution to the problem or prove that none exists in unsatisfiable instances.

Application 2: Weighted Max-CSP. The second setting we consider is that of computing a Max-CSP solution of an unsatisfiable constraint program. More specifically, this application takes as input an UNSAT CSP, and transforms this CSP into an optimization problem with reified (global) constraints. That is, given a partitioning of the constraints in a CSP into soft ($C_{\mathcal{S}}$) and hard constraints ($C_{\mathcal{H}}$), we compute an *optimal* solution to the following problem:

$$\begin{aligned} \textit{maximize} \quad & \sum_{s \in C_{\mathcal{S}}} w_s \cdot b_s \\ \textit{subject to} \quad & b_s \rightarrow s \quad \forall s \in C_{\mathcal{S}} \\ & C_{\mathcal{H}} \end{aligned}$$

In this experiment, we use the length of the string-representation of each constraint as weight, and we measure the time the solver takes to compute an *optimal* solution to the above problem. As discussed in the related works section, specialized algorithms for computing Max-CSP solutions exist, some of which allow for fine-grained penalties on violated constraints (De Givry 2023; Schiendorfer et al. 2018; van Hoeve, Pesant, et al. 2006). We consider an all-or-nothing setting where each constraint in the CSP is either turned on or off by half-reifying each constraint.

Application 3: Extracting an unsat core using assumption-based solving. Lazy-clause generation (LCG) solvers (Ohrimenko et al. 2009) like Chuffed and OR-Tools CP-SAT hybridize CP's constraint propagation with SAT solving. This allows such solvers to support solving with assumption variables inherited from the underlying SAT solver (Audemard, Lagniez, et al. 2013; Nadel and Ryvchin 2012). 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*. After solving, the solver can report a *sufficient set of assumptions* (a *core*) that causes the outcome to be UNSAT. Many practical implementations of MUS-extraction techniques heavily rely on this technique, for example, to warm-start the search of an MUS from a smaller core instead of starting from the full CSP (Nadel, Ryvchin, and Strichman 2014a; van Helvert 2021; Wieringa 2014). In this experiment, we find such a core by assuming all indicator variables to *true* before solving. We then measure both the time to prove the unsatisfiability and the size of the solver-reported core.

8.1 Benchmarks

In total, we use six benchmark sets to evaluate our reformulation. Set-game, RCPSP, and XCSP3 are well-known benchmarks from the literature, and are used for comparison in general classes of problems.

We also used 3 artificially generated benchmarks, with randomly generated instances, to investigate our reformulation in more detail. Specifically, we used Random-GCC, Random-AllDiff, and Random-Cumulative, which involve specific global constraints to cover the different cases (Cumulative as a total relation constraint, GlobalCardinality as a total function constraint, and AllDifferent as a general global constraint).

Set-game. We generate instances of the "set" game (Swearngin et al. 2011). In this game, the player is given a deck of n cards with various shapes. We distinguish the cards using k attributes (e.g., number of shapes, color, fill...) and each of these attributes can take v alternative values. The goal of the game is to find a "set" where for each attribute, the values are either all different or all the same. We model this puzzle with s integer variables x_i which represent the card-indices forming the set. The input of the model is a $n \times k$ matrix C. This leads to the following model after flattening, from which we generate 660 instances with varying s, v, n, and k.

$$\begin{aligned} &\textsc{AllDifferent}(x_1, x_2, \ldots, x_s) \\ b_j^{diff} \rightarrow \textsc{AllDifferent}(C[x_1, j], C[x_2, j] \ldots, C[x_s, j]) \quad &, \forall j = 1 \ldots k \\ b_j^{eq} \rightarrow \textsc{AllEqual}(C[x_1, j], C[x_2, j] \ldots, C[x_s, j]) \quad &, \forall j = 1 \ldots k \\ b_j^{diff} \vee b_j^{eq} \quad &, \forall j = 1 \ldots k \end{aligned}$$

RCPSP j60. We use 480 instances of the j60 class of Resource Constraint Project Scheduling Problem (RCPSP) instances from the PSPLib website[4]. Each of these instances is made unsatisfiable by limiting the makespan to 70% of the lower bound reported on the PSPLib website. The models consist of Cumulative constraints, and "$\leq$" constraints to enforce precedences between tasks. In the Max-CSP benchmarks, we treat the constraint enforcing the makespan to be better-than-optimal as a hard constraint, and all other constraints as soft.

XSCP3. We collected a total of 95 CSP instances from the UNSAT and optimization tracks of the XCSP3 competition series between 2022 and 2024. These models contain a variety of global constraints, including Table, NegativeTable, Regular, AllDifferent, and Cumulative constraints. The optimization instances are made unsatisfiable by enforcing the objective value to be better-than-optimal. Section C provides details on which global constraints are used in each of the instances.

Random-GCC.. We generate unsat CSPs containing $\textsc{GlobalCardinality}(X, V, N)$ constraints. We follow a similar approach to Bianco et al. (2019): given a pool of variables, we sample a subset of variables where each variable has a 50% probability of being picked. Lastly, we determine $|V| = |N|$ by sampling from a uniform distribution $[1..15]$ and sampling pairs of values and counts (v_i, n_i) such that the constraint can be satisfied. We range the number of global constraints from 2 to 5, resulting in a total of 200 unsatisfiable instances.

Random-AllDiff. Similar to the previous benchmark set, we construct unsatisfiable CSPs with randomly sampled AllDifferent constraints. We generate a pool of variables varying from 10 to 50 variables, and each variable has a 50% chance of being selected for each of the AllDifferent constraints. We range the number of constraints between 10 and 20, resulting in a total of 335 unsatisfiable instances.

Random-Cumulative. The last randomly generated benchmark set contains Cumulative constraints. We generate between 20 and 60 tasks with durations ranging from 5 to 10 time units and resource demands ranging

[4] psplib.com

from 1 to 5. We sample between 4 and 20 Cumulative constraints, where each task again has a 50% probability of being chosen for each constraint. This results in a benchmark set of 516 unsatisfiable instances.

8.2 Experimental Setup

We compare how rewriting half-reified global constraints performs to alternative strategies, such as decomposing or direct solver support for the half-reification:

- A decomposition of the implied global constraint into primitive constraints (**Decompose**).
- Our method (**Aux**) where all variables have been replaced.
- The version that uses a minimal number of auxiliary variables (**AuxMinimal**).
- Both variants of the previous two settings, while adding a dummy solution when the indicator variable is *false* (**AuxDummy** and **AuxMinDummy**).
- Native support (**Native**) for the half-reification of the global constraint if available.

As our reformulation introduces new variables in the problem, the branching heuristics of the solver are also influenced. Therefore, apart from running each benchmark with the **Default** branching order, we also employ two fixed branching orders: **orig-bv** and **bv-orig**. In **orig-bv**, we determine a fixed order in which all original variables are branched on (based on the input order into the model), before the solver branches on the Boolean indicator variables in the reformulation. Conversely, in **bv-orig**, we first branch on the indicator variables before branching on the original variables. Both orders branch on the maximum value in the current domain of the selected variable. In both search orders, the auxiliary variables are branched on last. The search orders **bv-orig** and **orig-bv** force the solver to only use the first and second case of the half-reification propagator (Eq. (1a) and Eq. (1b)) respectively.

We implemented our approaches on top of the CPMpy (Guns 2019) constraint modeling library v0.9.26 in Python 3.11. For solvers, we use OR-Tools CP-SAT (Perron and Furnon 2022) version 9.14, the Choco solver (Prud'homme et al. 2016) version 4.10.17, and the commercial solver CP-Optimizer v.22.1.1.0. As all code is written in Python, each solver is accessed through CPMpy and its Python interface (`pychoco` v.0.2.3 and `docplex` v.2.29.245). All solvers are set to work in single-threaded mode.

Of the solvers used, only Choco has native support for the half-reification of each global constraint in our benchmarks. Choco implements half-reification as described in Section 7. CP-Optimizer supports native reification for a subset of its global constraints. For our benchmarks, this is only for the AllDifferent constraint.

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 15 minutes. All code required to reproduce the experiments is available online: https://github.com/ML-KULeuven/Half-reified-global-constraints

8.3 EQ1: Reformulation vs Decomposing

In this first experimental question, we are interested in the runtime performance of the solver when using our reformulation rather than decomposing the half-reification, as is currently done in modeling systems.

We focus on Figures 2 to 6, using a default search order. Both for solving with compound constraints and solving a Max-CSP problem with half-reified constraints, our reformulation consistently outperforms the decomposition approach. For example, in RCPSP benchmark (Fig. 3), OR-Tools can solve 72 instances more using our reformulation compared to using the task-decomposition of the Cumulative constraint (Schutt, Feydy, Stuckey, and M. Wallace 2009). Moreover, the instances solved by both the reformulation and the decomposition are completed 10-100 times faster by the former. Similarly, for the Random-GCC benchmark with the CP-Optimizer solver (Fig. 4), the solver was able to solve *all* instances within 1s using our reformulation, while the instances using the decomposition of the GlobalCardinality constraint took up to 1000 times longer. This trend also continues for the Random-AllDiff and Random-Cumulative benchmarks (Figs. 5 and 6), where the decomposition approach

solves only a fraction of the instances compared to models using our reformulation. One exception is the results in Figure 4, showing the results for Choco on the Random-GCC benchmark. Here, the difference between our reformulation and the decomposition is less clear compared to the result for CP-Optimizer, as the propagation function implemented for GlobalCardinality in Choco has a weaker level of consistency compared to the propagator implemented in CP-Optimizer. Still, we can overall conclude that using our reformulation significantly speeds up solving compared to using the decomposition.

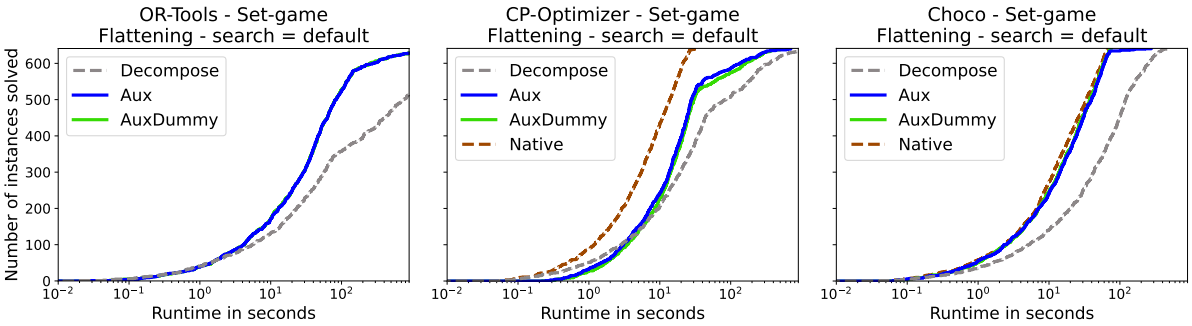

Fig. 2. Results for solving models with compound constraints after flattening using different solvers. OR-Tools does not support native reification of global constraints. **Aux** and **AuxDummy** overlap for both OR-Tools and Choco.

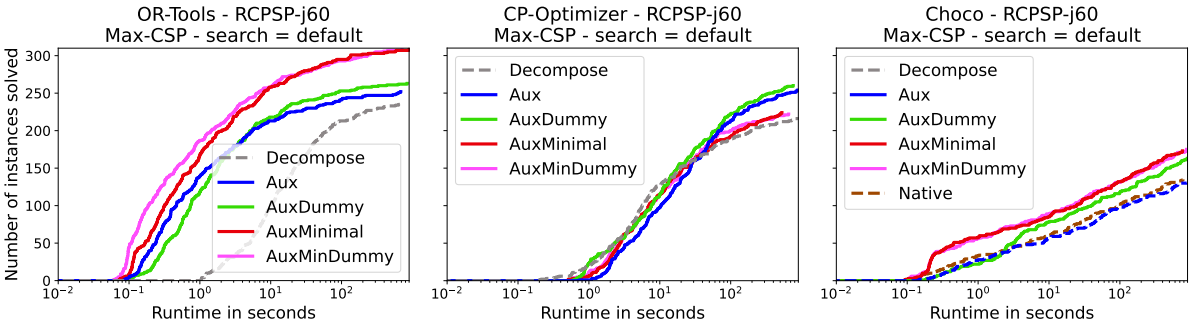

Fig. 3. Results for solving the Max-CSP problem on the RCPSP benchmark using different solvers.

OR-Tools does not support GlobalCardinality

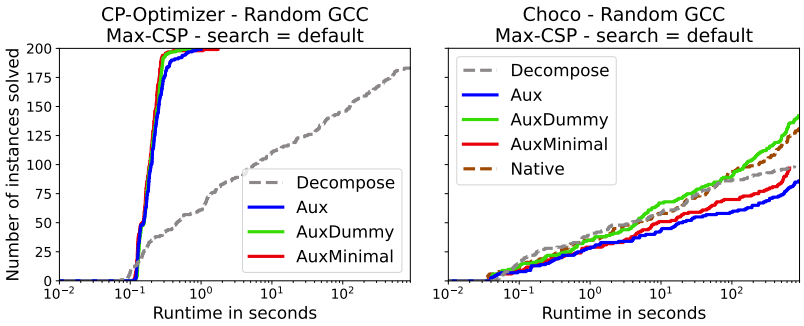

Fig. 4. Results for solving the Max-CSP problem on the Random-GCC benchmark using different solvers.

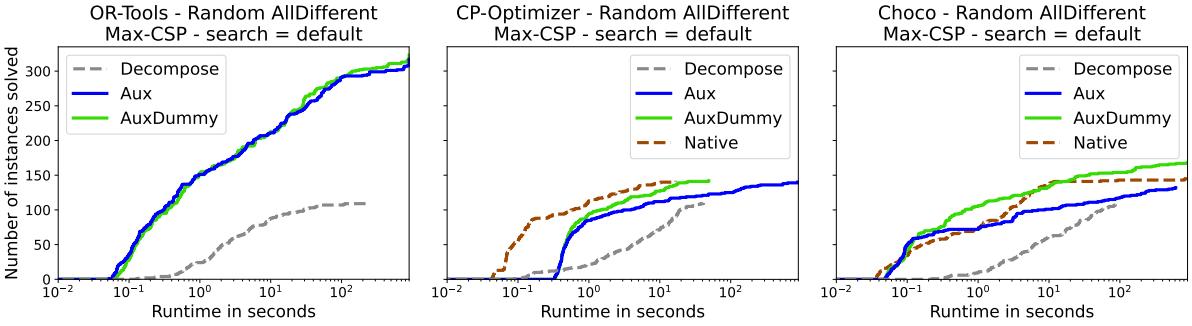

Fig. 5. Results for solving the Max-CSP problem on the Random-AllDiff benchmark using different solvers.

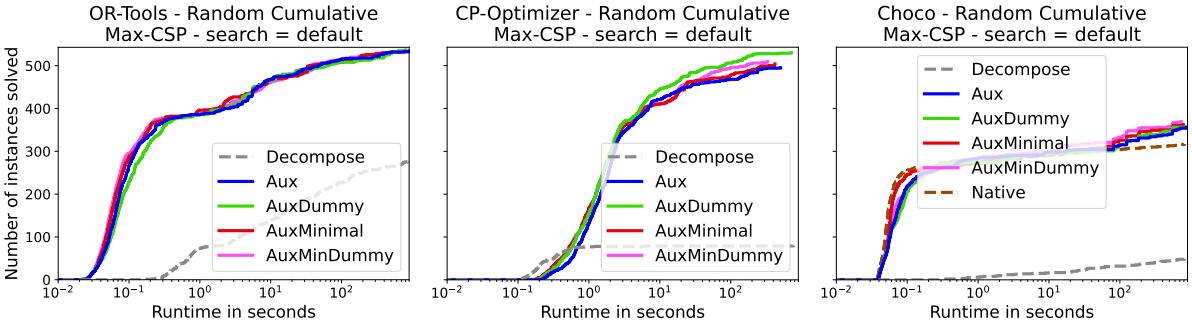

Fig. 6. Results for Solving a Max-CSP problem for different benchmarks and solvers

8.4 EQ2: Minimal Reformulation vs Replacing All Variables

In this second experimental question, we compare two versions of our reformulation: when replacing all variables with an auxiliary, and when replacing only the $\mathcal{Y}$ set for total relation constraints. We first focus on the results with a default search order in Figures 2 to 6. Here, we only notice a significant difference between replacing all variables and replacing only $\mathcal{Y}$ in the RCPSP benchmark when running with OR-Tools. Indeed, for this

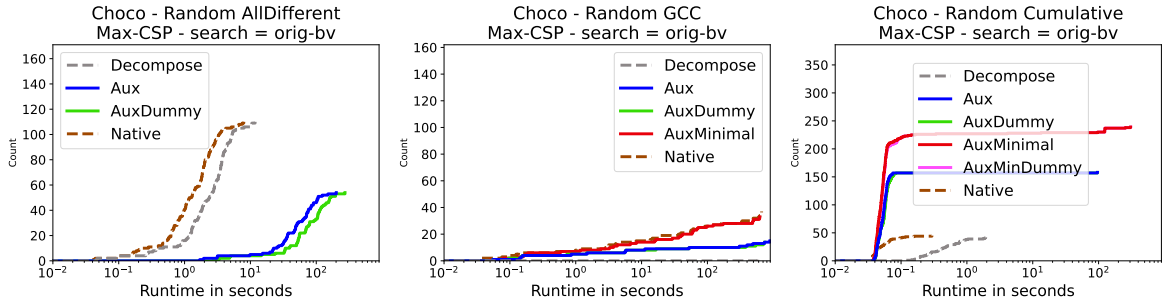

Fig. 7. Runtime for solving a Max-CSP problem for different benchmarks when branching on the original variables first. For the Random-GCC and Random-Cumulative, **Aux** and **AuxDummy** overlap, as does **AuxMinimal** and **AuxMinDummy**.

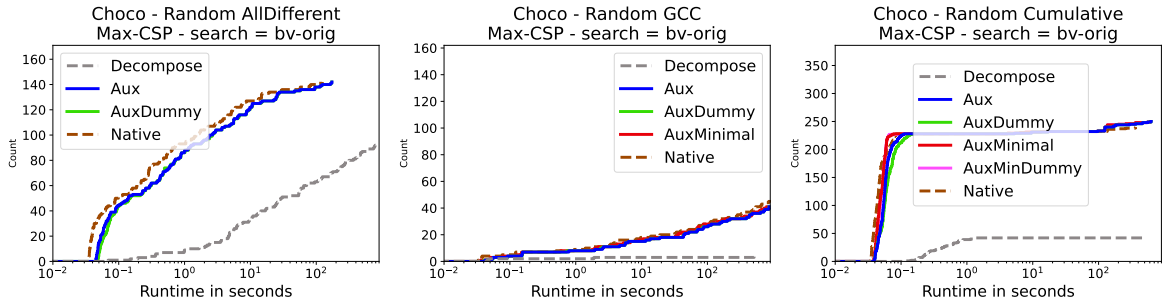

Fig. 8. Runtime for solving a Max-CSP problem for different benchmarks when branching on the indicator variable first. **Aux** and **AuxDummy** overlap for all benchmarks. For the Random-GCC and Random-Cumulative benchmarks, also minimal variants **AuxMinimal** and **AuxDummy** overlap with the other series.

benchmark, the solver is able to solve 308 instances using **AuxMinimal** compared to 252 instances when replacing all variables (**Aux**). For other solvers and benchmarks, the effect is less noticeable. Interestingly, for CP-Optimizer the performance of the reformulation actually degrades when using the minimal reformulation. Indeed, the minimal reformulation performance similar to the decomposition of the Cumulative constraint. As CP-Optimizer is a closed-source commercial solver, we cannot be certain why exactly this happens, though it is likely due to optimizations implemented to scheduling-specific constraints such as Cumulative.

We now turn our attention to Figures 7 and 8 where the branching order is fixed during search. Here, we do notice a clear difference between our minimal reformulation in both the Random-GCC and Random-Cumulative benchmarks. Indeed, for the Random-GCC benchmark, when branching on the original variables first (Fig. 7), the performance of our minimal reformulation is *identical* to the *native implementation*, and more than double the instances are solved using **AuxMinimal** over **Aux**. The same results hold for the Random-Cumulative benchmark, where replacing only the capacity variable results in an additional 81 instances solved to optimality.

8.5 EQ3: Forcing the Auxiliary Variables to a Dummy Solution

We now investigate the effect of adding an additional constraint that fixes the auxiliary variables during search, as described in Section 6.2. As expected, this does not have any effect on the runtime of the solver when fixing the search strategy, as shown in Figures 7 and 8. Indeed, as the auxiliary variables are pushed down in the search

tree by these fixed search orders, the addition of the dummy solution does not affect the solver's behaviour. Still, for the Random-AllDiff benchmark, we notice a slight decrease in performance, as the model contains more constraints when adding the dummy solution, and thus more work is done in each of the search nodes.

In contrast, when using the default search order of the solver, Choco clearly benefits from adding this dummy solution to the model in the AllDifferent benchmark (Fig. 5). This also holds for CP-Optimizer on that same benchmark, where the solver can solve a similar number of instances compared to the "basic" reformulation **Aux**, but it does so in a fraction of the time. Similarly, Choco benefits from adding a dummy solution for the auxiliary variables for the Random-GCC benchmark, as the propagator implemented in Choco has a relatively weak level of consistency compared to, for example, CP-Optimizer. Therefore, the solver requires many branching decisions on the auxiliary variables before finding a satisfying solution to the top-level GlobalCardinality constraint posted to the solver when using our reformulation **Aux**.

For all other solvers and benchmarks, the effect of **AuxDummy** over **Aux** is negligible, and we can conclude the default branching heuristics implemented in modern CP-solvers work well for avoiding branching on these auxiliary variables introduced by our reformulation.

8.6 EQ4: Reformulation vs Solver-Native Reification

In this fourth experiment question, we investigate the performance of our reformulation compared to when the solver natively supports the half-reification of the global constraint. As discussed in Section 7, our reformulation in general has a weaker level of consistency compared to the native propagator. Our experiments also confirm this, for example, in the Set-game in Figure 2, the native reformulation consistently outperforms our reformulation for both CP-Optimizer and Choco. However, across many benchmarks, the solver-native reification performs similarly to, or even worse than, our reformulation. Consider, for example, the Random-AllDiff benchmark in Figure 5. Here, using our best reformulation (green series) results in more instances solved compared to using the solver-native reified AllDifferent constraints. The main reason for this behaviour is the way Choco implements its checker for the AllDifferent constraint. As discussed in Section 7.3, the checker for this particular constraint is only enabled when all variables are instantiated. Hence, failure of the reified constraint is sometimes detected earlier using our reformulation compared to the native implementation. We notice similar behaviour for the Random-GCC benchmark in Figure 4. Here, our reformulation slightly outperforms Choco's native implementation for reified GlobalCardinality constraints. This is because their checker does not detect when the counting variables do not sum to the number of variables in the constraint. As discussed in Section 7, this means our reformulation can have an equal or better propagation strength compared to the native half-reified propagator.

We now compare the propagation strength of our reformulation to a solver-level implementation by fixing the search order. Consider the results for the AllDifferent benchmark while branching on the original decision variables first (Fig. 7). Here, we notice that both the native implementation and even the decomposition clearly outperform our reformulation. This is to be expected as our reformulation does not model the negation of the global constraint. As discussed in Section 7, this results in fewer cases where the solver can detect failure of the reified constraint. Conversely, the binary decomposition of the AllDifferent constraint does model the negation. Indeed, the AllDifferent constraint is only unsatisfied if at least one of the '$\neq$' constraints in the decomposition is unsatisfied.

In Figure 8, we first branch on the indicator variables in the half-reifications. Therefore, we evaluate how the first case of the half-reified propagator (Equation (1a)) compares to our half-reification. As expected from the discussion in Section 7, the solve-time using our reformulation is almost identical to using Choco's native half-reification propagators. Any differences in the plots are due to more constraints being posted in the solver, and the small overhead introduced when reformulating the global constraint.

When comparing across solvers, we clearly see the advantage of using a model-level reformulation. Consider, for instance, the Random-Cumulative benchmark. Here, Choco was able to solve 317 instances with the native reformulation, whereas CP-Optimizer and OR-Tools were able to solve 537 and 530 instances using our reformulation. This is in contrast with only 277 and 80 instances solved using the decomposition of Cumulative. Hence, by using our reformulation, these problems can now be solved faster using CP-Optimizer or OR-Tools compared to what was possible before, even when using a solver with native support for half-reified global constraints.

8.7 EQ5: Effect on UNSAT-Core Extraction

In this final experiment, we investigate our reformulation rule in the context of extracting an UNSAT core using a lazy-clause generation solver such as OR-Tools. Such an unsat core is often used in the context of MUS-extraction algorithms where (CP-)solvers are used as an *oracle* (Bleukx, Guns, et al. 2024; Wieringa 2014). Naturally, these algorithms benefit from a fast runtime of the solver, but also the *size* of the solver-reported core can have an impact on the overall runtime of the "main" MUS-algorithm. We now investigate these metrics in detail when using our reformulation.

Figure 9 shows that our reformulations also outperform the current approach of decomposing the half-reified global constraint. While this is less noticeable in the XCSP3 benchmark, for both Random-AllDiff and Random-Cumulative, a clear difference between **Aux** and **Decompose** is visible. Indeed, for the Random-AllDiff benchmark, OR-Tools was able to solve all 550 instances within 5s of runtime, using our reformulation, whereas the binary decomposition of AllDifferent resulted in only 110 instances solved within the time limit of 900s. Interestingly, some instances are solved in the Random-Cumulative benchmark with the decomposition but are not solved using any of our reformulations. This is likely due to optimizations in the `presolve` functionality of OR-Tools, where the value of assumption literals are not taken into account at this time of the solving process.

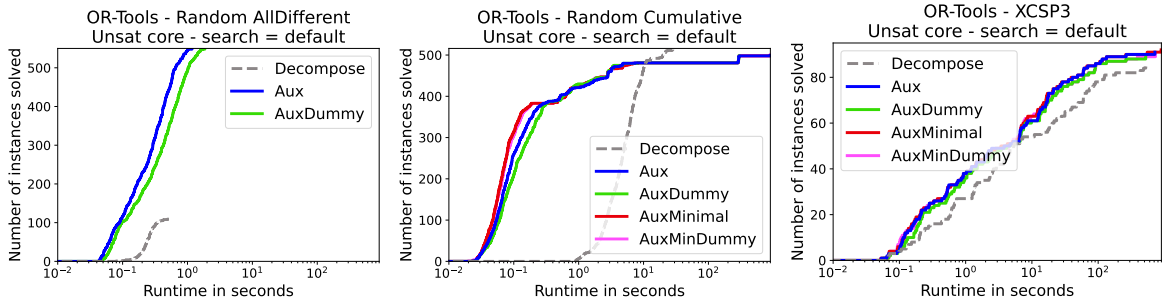

Fig. 9. Runtime for extracting an unsatisfiable core on different benchmarks.

For the second metric, we compare the size of the solver-reported core after it determined the model was UNSAT. In general, a solver-reported core is a subset of the "assumption variables" set by the user (Marques-Silva, Lynce, et al. 2021). In our setting, this means the solver-reported core consists of a set of indicator variables from the half-reification constraints. The exact set of assumption variables that end up in the core is influenced by (1) the clause-learning mechanism of the solver and (2) the constraints that are propagated before reaching a conflict in the search tree. This second aspect is influenced by our reformulation. Indeed, as the solver can now use its global constraint propagator, the number of constraint propagation calls required to reach a conflict can be substantially reduced. The effect is clearly visible in Table 2 where we report the size of the solver-reported core after the solve-call has finished. For the Random-AllDiff benchmark, the effect is most noticeable, where the solver-reported core consists of only 1.45 constraints on average. In contrast, the solver reported a core of size

9.87 when using the binary decomposition of AllDifferent. Similarly, for the Random-Cumulative and XCSP3 benchmarks, the size of the core is reduced from 7.46 to 2.33 and from 112.95 to 99.60 constraints, respectively.

Overall, we can conclude that our reformulation has a positive effect on both the runtime of the solver and on the size of the solver-reported core when solving under assumptions.

Table 2. Size of solver-reported core after solving under assumptions

	Random-AllDiff		Random-Cumulative		XCSP3	
	mean	std	mean	std	mean	std
Decompose	9.87	2.59	7.46	3.06	112.95	201.27
Aux	**1.45**	1.76	2.35	4.19	107.26	164.58
AuxMinimal	-	-	**2.33**	4.16	107.17	164.59
AuxDummy	1.49	1.94	2.36	4.23	99.68	147.42
AuxMinDummy	-	-	**2.33**	4.17	**99.60**	147.44

9 Conclusion and Future Work

We introduced and formalized several simple model-level reformulations for half-reified global constraints. These reformulations allow for solving any model with half-reified global constraints by any CP-solver that supports the top-level (non-reified) global constraint. Crucially, our reformulations avoid decomposing the global constraint and instead rewrite it by introducing auxiliary variables and a reified channeling constraint. Therefore, we can still use the efficient global constraints found in CP solvers during search, when the Boolean indicator variable of the half-reification is assigned to *true*.

In an extensive experimental evaluation covering several benchmarks and applications of half-reified global constraints, we have shown that our reformulation speeds up the solving process by several orders of magnitude compared to reifying the decomposition. Moreover, for some total-function and total-relation constraints, we have shown that our reformulation, both theoretically and empirically, matches the native half-reification propagator as proposed by Feydy et al. (2011) and implemented in the Choco solver. Additionally, when the solver implements half-reification using a checking algorithm instead of using the constraint propagator, our model-level reformulation can even *outperform* a solver-level implementation of half-reified global constraints.

Our work enables the use of half-reified global constraints for any solver supporting that global constraint. Hence, XCP-related techniques, or solving CSPs with nested global constraints, can now make efficient use of any existing solver rather than relying only on solvers that natively support reified global constraints. Moreover, due to significant runtime improvements, techniques such as MUS-extraction that make heavy use of half-reified constraints can now be applied to much larger and complex models that contain global constraints.

The main downside of our model-level reformulation compared to a solver-level specialized propagator is that our general reformulation cannot filter the Boolean indicator variable to be *false* when the reified constraint fails. While this is irrelevant for some applications (e.g., unsat core extraction using lazy-clause generation solvers), we believe future improvements can further enhance this aspect of our reformulation. As mentioned in the experiments, the decomposition of global constraints typically *does* model the negation of the global constraint as well. Hence, we could combine (a subset of) a decomposition of a global constraint with our reformulation. For example, while posting the full n^2 binary decomposition of a AllDifferent constraints is unwanted, we can reify a subset of the "$\neq$" constraints and post this alongside our reformulation. Such a decomposition would act as redundant constraints in the model, but can improve the runtime of the solver. Still, this would require fine-grained information about both the global constraint in question and the decomposition used. In contrast, the reformulations we proposed are generic and work for any global constraint in the global constraint catalog.

Acknowledgments

This research is partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt) and by the Research Foundation-Flanders (FWO-Vlaanderen, G020525N).

References

A. Aggoun and N. Beldiceanu. 1992. "Extending CHIP in Order to Solve Complex Scheduling and Placement Problems." In: *JFPL'92, 1ères Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*. Ed. by J. Delahaye, P. Devienne, P. Mathieu, and P. Yim, 51.

Ö. Akgün, A. M. Frisch, I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. 2022. "Conjure: Automatic Generation of Constraint Models from Problem Specifications." *Artif. Intell.*, 310, 103751. DOI: 10.1016/J.ARTINT.2022.103751.

G. Audemard, J. Lagniez, and L. Simon. 2013. "Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction." In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings* (Lecture Notes in Computer Science). Ed. by M. Järvisalo and A. Van Gelder. Vol. 7962. Springer, 309–317. DOI: 10.1007/978-3-642-39071-5_23.

G. Audemard, C. Lecoutre, and E. Lonca. 2024. "Proceedings of the 2024 XCSP3 Competition." *CoRR*, abs/2412.00117.

P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. 2006. "Constraint-Based Scheduling and Planning." In: *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Ed. by F. Rossi, P. van Beek, and T. Walsh. Elsevier, 761–799. DOI: 10.1016/S1574-6526(06)80026-X.

N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. Feb. 2012. *On the Reification of Global Constraints*. Technical report T2012:02. TASC team (INRIA/CNRS), SCS, and Uppsala University, Department of Information Technology, (Feb. 2012).

N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. 2013. "On the Reification of Global Constraints." *Constraints An Int. J.*, 18, 1, 1–6. DOI: 10.1007/S10601-012-9132-0.

N. Beldiceanu, M. Carlsson, and J.-X. Rampon. 2012. *Global Constraint Catalog, (Revision A)*. (2012).

G. Belov, P. J. Stuckey, G. Tack, and M. Wallace. 2016. "Improved Linearization of Constraint Programming Models." In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings* (Lecture Notes in Computer Science). Ed. by M. Rueher. Vol. 9892. Springer, 49–65. DOI: 10.1007/978-3-319-44953-1_4.

C. Bessiere. 2006. "Constraint Propagation." In: *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Ed. by F. Rossi, P. van Beek, and T. Walsh. Elsevier, 29–83. DOI: 10.1016/S1574-6526(06)80007-6.

C. Bessière. 1994. "Arc-Consistency and Arc-Consistency Again." In: 1. Vol. 65, 179–190.

C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. 2006. "Filtering Algorithms for the NValue Constraint." *Constraints An Int. J.*, 11, 4, 271–293. DOI: 10.1007/S10601-006-9001-9.

C. Bessiere, G. Katsirelos, N. Narodytska, and T. Walsh. 2009. "Circuit Complexity and Decompositions of Global Constraints." In: *IJCAI*, 412–418.

C. Bessière and P. Van Hentenryck. 2003. "To Be or Not to Be ... a Global Constraint." In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings* (Lecture Notes in Computer Science). Ed. by F. Rossi. Vol. 2833. Springer, 789–794. DOI: 10.1007/978-3-540-45193-8_54.

G. L. Bianco, X. Lorca, C. Truchet, and G. Pesant. 2019. "Revisiting Counting Solutions for the Global Cardinality Constraint." *J. Artif. Intell. Res.*, 66, 411–441. DOI: 10.1613/JAIR.1.11325.

A. Biere, M. Heule, H. van Maaren, and T. Walsh, (Eds.) . 2021. *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. IOS Press. ISBN: 978-1-64368-160-3. DOI: 10.3233/FAIA336.

G. Björdal, P. Flener, J. Pearson, P. J. Stuckey, and G. Tack. 2020. "Solving Satisfaction Problems Using Large-Neighbourhood Search." In: *International Conference on Principles and Practice of Constraint Programming*. Springer, 55–71.

I. Bleukx, R. Boumazouza, T. Guns, N. Laage, and G. Povéda. 2025. "Modeling and Explaining an Industrial Workforce Allocation and Scheduling Problem." In: *31st International Conference on Principles and Practice of Constraint Programming, CP 2025, August 10-15, 2025, Glasgow, Scotland* (LIPIcs). Ed. by M. G. de la Banda. Vol. 340. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:24. DOI: 10.4230/LIPICS.CP.2025.6.

[SW] I. Bleukx, T. Guns, and D. Tsouros, *Explainable Constraint Solving: A Hands-On Tutorial* version v1.0, Feb. 2024. DOI: 10.5281/zenodo.10694140, URL: https://doi.org/10.5281/zenodo.10694140.

S. Bourdais, P. Galinier, and G. Pesant. 2003. "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* (Lecture Notes in Computer Science). Ed. by F. Rossi. Vol. 2833. Springer, 153–167. DOI: 10.1007/978-3-540-45193-8_11.

F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim. 2008. "A New Constraint Programming Approach for the Orthogonal Packing Problem." *Comput. Oper. Res.*, 35, 3, 944–959. DOI: 10.1016/J.COR.2006.05.012.

S. De Givry. 2023. "toulbar2, an Exact Cost Function Network Solver." In: *24ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision ROADEF 2023*.

C. Dejemeppe. 2016. "Constraint Programming Algorithms and Models for Scheduling Applications." Ph.D. Dissertation. Catholic University of Louvain, Louvain-la-Neuve, Belgium. https://hdl.handle.net/2078.1/178078.

F. Fages and S. Soliman. Oct. 2012. *Reifying Global Constraints*. Research Report RR-8084. INRIA, (Oct. 2012), 18. https://inria.hal.science/hal-00737768.

T. Feydy, Z. Somogyi, and P. J. Stuckey. 2011. "Half Reification and Flattening." In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings* (Lecture Notes in Computer Science). Ed. by J. H. Lee. Vol. 6876. Springer, 286–301. DOI: 10.1007/978-3-642-23786-7_23.

M. Flippo, K. Sidorov, I. Marijnissen, J. Smits, and E. Demirovic. 2024. "A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers." In: *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain* (LIPIcs). Ed. by P. Shaw. Vol. 307. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:20. DOI: 10.4230/LIPICS.CP.2024.11.

A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, and I. Miguel. 2008. "Essence : A Constraint Language for Specifying Combinatorial Problems." *Constraints An Int. J.*, 13, 3, 268–306. DOI: 10.1007/S10601-008-9047-Y.

A. M. Frisch and P. J. Stuckey. 2009. "The Proper Treatment of Undefinedness in Constraint Languages." In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings* (Lecture Notes in Computer Science). Ed. by I. P. Gent. Vol. 5732. Springer, 367–382. DOI: 10.1007/978-3-642-04244-7_30.

I. P. Gent, C. Jefferson, and I. Miguel. 2006. "Minion: A Fast Scalable Constraint Solver." In: *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* (Frontiers in Artificial Intelligence and Applications). Ed. by G. Brewka, S. Coradeschi, A. Perini, and P. Traverso. Vol. 141. IOS Press, 98–102. http://www.booksonline.iospress.nl/Content/View.aspx?piid=1654.

I. P. Gent, K. E. Petrie, and J. Puget. 2006. "Symmetry in Constraint Programming." In: *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Elsevier, 329–376.

T. Guns. 2019. "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)*. Vol. 19.

A. Ignatiev, A. Previti, M. H. Liffiton, and J. Marques-Silva. 2015. "Smallest MUS Extraction with Minimal Hitting Set Dualization." In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings* (Lecture Notes in Computer Science). Ed. by G. Pesant. Vol. 9255. Springer, 173–182. DOI: 10.1007/978-3-319-23219-5_13.

M. Iori, V. L. de Lima, S. Martello, F. K. Miyazawa, and M. Monaci. 2021. "Exact Solution Techniques for Two-Dimensional Cutting and Packing." *Eur. J. Oper. Res.*, 289, 2, 399–415. DOI: 10.1016/J.EJOR.2020.06.050.

C. Jefferson, N. C. A. Moore, P. Nightingale, and K. E. Petrie. 2010. "Implementing Logical Connectives in Constraint Programming." *Artif. Intell.*, 174, 16-17, 1407–1429. DOI: 10.1016/J.ARTINT.2010.07.001.

U. Junker. 2001. "QuickXplain: Conflict Detection for Arbitrary Constraint Propagation Algorithms." In: *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*. Vol. 4. Citeseer.

L. G. Kaya and J. N. Hooker. 2006a. "A Filter for the Circuit Constraint." In: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by F. Benhamou. Vol. 4204. Springer, 706–710. DOI: 10.1007/11889205_55.

L. G. Kaya and J. N. Hooker. 2006b. "A Filter for the Circuit Constraint." In: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by F. Benhamou. Vol. 4204. Springer, 706–710. DOI: 10.1007/11889205_55.

M. Z. Lagerkvist and C. Schulte. 2009. "Propagator Groups." In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings* (Lecture Notes in Computer Science). Ed. by I. P. Gent. Vol. 5732. Springer, 524–538. DOI: 10.1007/978-3-642-04244-7_42.

J. Larrosa, P. Meseguer, and T. Schiex. 1999. "Maintaining Reversible DAC for Max-CSP." *Artif. Intell.*, 107, 1, 149–163. DOI: 10.1016/S0004-3702(98)00108-8.

M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. 2016. "Fast, Flexible MUS Enumeration." *Constraints An Int. J.*, 21, 2, 223–250. DOI: 10.1007/S10601-015-9183-0.

J. Marques-Silva. 2010. "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*. IEEE Computer Society, 9–14. DOI: 10.1109/ISMVL.2010.11.

J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. 2013. "On Computing Minimal Correction Subsets." In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. Ed. by F. Rossi. IJCAI/AAAI, 615–622. http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6922.

J. Marques-Silva, I. Lynce, and S. Malik. 2021. "Conflict-Driven Clause Learning SAT Solvers." In: *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. IOS Press, 133–182. DOI: 10.3233/FAIA200987.

C. Mears, M. G. de la Banda, M. Wallace, and B. Demoen. 2015. "A Method for Detecting Symmetries in Constraint Models and Its Generalisation." *Constraints An Int. J.*, 20, 2, 235–273. DOI: 10.1007/S10601-014-9175-5.

C. Mencía, A. Ignatiev, A. Previti, and J. Marques-Silva. 2016. "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*. Springer, 342–360.

A. Nadel and V. Ryvchin. 2012. "Efficient SAT Solving under Assumptions." In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. Springer, 242–255. DOI: 10.1007/978-3-642-31612-8_19.

A. Nadel, V. Ryvchin, and O. Strichman. 2014a. "Accelerated Deletion-Based Extraction of Minimal Unsatisfiable Cores." *J. Satisf. Boolean Model. Comput.*, 9, 1, 27–51. DOI: 10.3233/SAT190100.

A. Nadel, V. Ryvchin, and O. Strichman. 2014b. "Ultimately Incremental SAT." In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings* (Lecture Notes in Computer Science). Ed. by C. Sinz and U. Egly. Vol. 8561. Springer, 206–218. DOI: 10.1007/978-3-319-09284-3_16.

N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. 2007. "MiniZinc: Towards a Standard CP Modelling Language." In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings* (Lecture Notes in Computer Science). Ed. by C. Bessiere. Vol. 4741. Springer, 529–543. DOI: 10.1007/978-3-540-74970-7_38.

P. Nightingale, P. Spracklen, and I. Miguel. 2015. "Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row." In: *CP* (Lecture Notes in Computer Science). Vol. 9255. Springer, 330–340.

O. Ohrimenko, P. J. Stuckey, and M. Codish. 2009. "Propagation via Lazy Clause Generation." *Constraints An Int. J.*, 14, 3, 357–391. DOI: 10.1007/S10601-008-9064-X.

[SW] L. Perron and V. Furnon, *OR-Tools* version v9.6, Nov. 2022. Google. URL: https://developers.google.com/optimization/.

D. Pisinger and M. Sigurd. 2007. "Using Decomposition Techniques and Constraint Programming for Solving the Two-Dimensional Bin-Packing Problem." *INFORMS J. Comput.*, 19, 1, 36–51. DOI: 10.1287/IJOC.1060.0181.

D. A. Plaisted and S. Greenbaum. 1986. "A Structure-Preserving Clause Form Translation." *J. Symb. Comput.*, 2, 3, 293–304. DOI: 10.1016/S0747-7171(86)80028-1.

C. Prud'homme, J.-G. Fages, and X. Lorca. 2016. "Choco Solver Documentation." *TASC, INRIA Rennes, LINA CNRS UMR*, 6241, 102–106.

J. Régin. 2011. "Using Hard Constraints for Representing Soft Constraints." In: *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* (Lecture Notes in Computer Science). Ed. by T. Achterberg and J. C. Beck. Vol. 6697. Springer, 176–189. DOI: 10.1007/978-3-642-21311-3_17.

A. Rendl, I. P. Gent, and I. Miguel. 2008. "Eliminating Common Subexpressions during Flattening." *URL: http://www-circa. mcs. st-and. ac. uk/Preprints/ERCIMCSE08. pdf.*

F. Rossi, P. van Beek, and T. Walsh, (Eds.) . 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Elsevier. ISBN: 978-0-444-52726-4. https://www.sciencedirect.com/science/bookseries/15746526/2.

A. Schiendorfer, A. Knapp, G. Anders, and W. Reif. 2018. "MiniBrass: Soft Constraints for MiniZinc." *Constraints An Int. J.*, 23, 4, 403–450. DOI: 10.1007/S10601-018-9289-2.

C. Schulte. 2000. "Programming Deep Concurrent Constraint Combinators." In: *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings* (Lecture Notes in Computer Science). Ed. by E. Pontelli and V. S. Costa. Vol. 1753. Springer, 215–229. DOI: 10.1007/3-540-46584-7_15.

C. Schulte and P. J. Stuckey. 2008. "Efficient Constraint Propagation Engines." *ACM Trans. Program. Lang. Syst.*, 31, 1, 2:1–2:43. DOI: 10.1145/1452044.1452046.

C. Schulte, G. Tack, and M. Z. Lagerkvist. 2010. "Modeling and Programming with Gecode." *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1.

A. Schutt, T. Feydy, P. J. Stuckey, and M. Wallace. 2009. "Why Cumulative Decomposition Is Not as Bad as It Sounds." In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings* (Lecture Notes in Computer Science). Ed. by I. P. Gent. Vol. 5732. Springer, 746–761. DOI: 10.1007/978-3-642-04244-7_58.

A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. 2011. "Explaining the Cumulative Propagator." *Constraints An Int. J.*, 16, 3, 250–282. DOI: 10.1007/S10601-010-9103-2.

A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. 2010. "Solving the Resource Constrained Project Scheduling Problem with Generalized Precedences by Lazy Clause Generation." *CoRR*, abs/1009.0347. http://arxiv.org/abs/1009.0347 arXiv: 1009.0347.

P. Shaw. 1998. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems." In: *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings* (Lecture Notes in Computer Science). Ed. by M. J. Maher and J. Puget. Vol. 1520. Springer, 417–431. DOI: 10.1007/3-540-49481-2_30.

P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer. 2014. "The MiniZinc Challenge 2008-2013." *AI Mag.*, 35, 2, 55–60.

A. Swearngin, B. Y. Choueiry, and E. C. Freuder. 2011. "A Reformulation Strategy for Multi-Dimensional CSPs: The Case Study of the SET Game." In: *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona,*

Catalonia, Spain, July 17-18, 2011. Ed. by M. R. Genesereth and P. Z. Revesz. AAAI. http://www.aaai.org/ocs/index.php/SARA/SARA11/paper/view/4251.
E. Tsang, J. Ford, P. Mills, R. Bradwell, R. Williams, and P. Scott. 2004. "ZDC-Rostering: A Personnel Scheduling System Based On Constraint Programming." *Technical Report*.
E. P. K. Tsang. 2003a. "Constraint Based Scheduling: Applying Constraint Programming to Scheduling Problems." *J. Sched.*, 6, 4, 413–414. DOI: 10.1023/A:1024016929283.
E. P. K. Tsang. 2003b. "Constraint Based Scheduling: Applying Constraint Programming to Scheduling Problems." *J. Sched.*, 6, 4, 413–414. DOI: 10.1023/A:1024016929283.
B. van Helvert. 2021. "A Solver Agnostic Incremental Machine Reasoning Interface." Master's thesis. Eindhoven University of Technology.
W.-J. van Hoeve. 2001. "The AllDifferent Constraint: A Survey." *CoRR*, cs.PL/0105015. https://arxiv.org/abs/cs/0105015.
W.-J. van Hoeve and I. Katriel. 2006. "Global Constraints." In: *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Ed. by F. Rossi, P. van Beek, and T. Walsh. Elsevier, 169–208. DOI: 10.1016/S1574-6526(06)80010-6.
W.-J. van Hoeve, G. Pesant, and L. Rousseau. 2006. "On Global Warming: Flow-Based Soft Global Constraints." *J. Heuristics*, 12, 4-5, 347–373. DOI: 10.1007/S10732-006-6550-4.
S. L. Vasileiou, A. Previti, and W. Yeoh. 2021. "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*. AAAI Press, 6514–6521. DOI: 10.1609/AAAI.V35I7.16807.
M. Wallace and M. Wallace. 2020. "Constraint Classes and Solvers." *Building Decision Support Systems: Using MiniZinc*, 85–111.
S. Wieringa. Mar. 2014. "Incremental Satisfiability Solving and Its Applications." PhD thesis. Aalto University, (Mar. 2014).

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. → means the solver has support for the half-reification of the global constraint.

Table 3. Solver support for different global constraints.

Solver	OR-Tools	Choco	CP-Optimizer
AllDifferent	✓	✓, →	✓, →
Cumulative	✓	✓, →	✓
Circuit	✓	✓, →	✓
GlobalCardinality		✓, →	✓

B Tabular Results

Table 4 summarizes the number of finished instances for each of the benchmarks, in each of the application settings. Note that not all reformulation variants are compatible for all global constraint types (e.g., AuxMinimal is not applicable for AllDifferent. Additionally, the Set-game benchmark is only applicable for the setting of "just solving" the model, as the half-reified global constraints arise naturally in the model during flattening.

Table 5 summarizes the number of finished instances by Choco for each of the benchmarks, for all variants of our reformulation, and different search orders.

Table 4. Number of finished instances for all benchmarks and solvers with default search order

Solver	Variant	Set Flattening	RCPSP MaxCSP	GCC MaxCSP	Alldiff MaxCSP	Alldiff Assump	Cumulative MaxCSP	Cumulative Assump	XCSP Assump
OR-Tools	Decompose	513	236	-	110	110	277	**513**	85
	Aux	628	252	-	317	**550**	534	499	91
	AuxDummy	629	263	-	**323**	**550**	**537**	498	89
	AuxMinimal	-	**308**	-	-	-	534	499	**92**
	AuxMinDummy	-	**308**	-	-	-	533	501	90
CP-Optimizer	Decompose	633	218	184	110	-	80	-	-
	Aux	640	254	**200**	140	-	495	-	-
	AuxDummy	640	260	**200**	142	-	530	-	-
	AuxMinimal	-	224	**200**	-	-	504	-	-
	AuxMinDummy	-	222	-	-	-	509	-	-
	Native	**641**	-	-	142	-	-	-	-
Choco	Decompose	**641**	0	99	110	-	49	-	-
	Aux	**641**	131	86	132	-	356	-	-
	AuxDummy	**641**	163	142	168	-	357	-	-
	AuxMinimal	-	173	97	-	-	362	-	-
	AuxMinDummy	-	175	-	-	-	369	-	-
	Native	**641**	135	132	146	-	317	-	-

Table 5. Number of finished instances by Choco for all benchmarks with different search orders

Solver	Search order	Variant	Random-AllDiff MaxCSP	Random-Cumulative MaxCSP	Random-GCC MaxCSP
Choco	bv-orig	Decompose	94	43	4
		Aux	**142**	**250**	41
		AuxDummy	**142**	**250**	41
		AuxMinDummy	-	**250**	-
		AuxMinimal	-	**250**	41
		Native	**142**	240	46
	orig-bv	Decompose	**110**	42	1
		Aux	54	158	15
		AuxDummy	54	158	14
		AuxMinDummy	-	**239**	-
		AuxMinimal	-	**239**	33
		Native	**110**	45	**37**

C Overview of Instances in XCSP3 Benchmark

Received 02 December 2025; accepted 30 June 2026

Table 6. Overview of constraints present in XCSP3 instances

Instance	AllDifferent	Cumulative	Inverse	InDomain	NegativeTable	NoOverlap	NoOverlap2D	Regular	ShortTable	Table
AntimagicSquare-03_c23	2	0	0	0	0	0	0	0	0	0
AztecDiamondSym-03_c24	0	0	0	0	0	0	0	0	24	0
BeerJugs-dec-01_c23	0	0	0	0	0	0	0	0	0	70
BeerJugs-dec-07_c23	0	0	0	0	0	0	0	0	0	70
BinPacking-n1c1w1a_c24	0	0	0	232	0	0	0	0	0	0
BinPacking-table-n1c1w1a_c24	0	0	0	232	0	0	0	0	0	29
BinPacking2-n1c1w1a_c24	0	0	0	0	0	0	0	0	0	1
Cargo-02-0s-1139_c24	0	3	0	0	0	0	1	0	0	0
Cargo-04-1s-626_c24	0	3	0	0	0	0	1	0	0	0
Cargo-05-1s-954_c24	0	3	0	0	0	0	1	0	0	0
Cargo-06-1s-3927_c24	0	3	0	0	0	0	1	0	0	0
Cargo-07-1s-133_c24	0	3	0	0	0	0	1	0	0	0
Cargo-09-1s-18_c24	0	3	0	0	0	0	1	0	0	0
Cargo-10-15966f-2060_c24	0	3	0	0	0	0	1	0	0	0
Cargo-22_c24	0	3	0	0	0	0	1	0	0	0
CarpetCutting-test01_c23	0	2	0	3	0	0	1	0	0	0
CarpetCutting-test02_c23	0	2	0	2	0	0	1	0	0	0
CarpetCutting-test03_c23	0	2	0	3	0	0	1	0	0	1
CarpetCutting-test04_c23	0	2	0	5	0	0	1	0	0	1
CarpetCutting-test05_c23	0	2	0	5	0	0	1	0	0	1
CarpetCutting-test10_c23	0	2	0	13	0	0	1	0	0	1
CarpetCutting-test11_c23	0	2	0	3	0	0	1	0	0	0
CarpetCutting-test16_c23	0	2	0	5	0	0	1	0	0	1
Charlotte-06-0_c24	31	0	0	36	0	0	0	0	237	0
Charlotte-06-1_c24	31	0	0	36	0	0	0	0	237	0
Charlotte-06-2_c24	31	0	0	36	0	0	0	0	237	0
FastMatrixMultiplication-2-2-2-04	0	0	0	0	0	0	0	0	0	0
FastMatrixMultiplication-2-2-2-05	0	0	0	0	0	0	0	0	0	0
FastMatrixMultiplication-table-2-2-2-04	0	0	0	0	0	0	0	0	256	0
FastMatrixMultiplication-table-2-2-2-05	0	0	0	0	0	0	0	0	320	0
FoolSolitaire-dec2-3-0	0	0	0	0	29	0	0	0	30	1
HCPizza-10-10-2-6-00_c23	0	0	0	428	0	0	0	0	0	428
HCPizza-12-12-2-6-00_c23	0	0	0	543	0	0	0	0	0	543
HCPizza-12-12-2-6-01_c23	0	0	0	618	0	0	0	0	0	618
HCPizza-12-12-2-6-02_c23	0	0	0	502	0	0	0	0	0	502
HCPizza-15-15-2-7-00_c23	0	0	0	1211	0	0	0	0	0	1211
HCPizza-15-15-2-7-01_c23	0	0	0	1276	0	0	0	0	0	1276
HCPizza-15-15-2-7-02_c23	0	0	0	1136	0	0	0	0	0	1136
HCPizza-20-20-2-8-00_c23	0	0	0	3635	0	0	0	0	0	3635
HCPizza-20-20-2-8-01_c23	0	0	0	3834	0	0	0	0	0	3834
HCPizza-20-20-2-8-02_c23	0	0	0	3650	0	0	0	0	0	3650
Hsp-10405_c23	1	0	0	0	0	1	0	0	0	1
Hsp-11406_c23	1	0	0	0	0	1	0	0	0	1
Hsp-12407_c23	1	0	0	0	0	1	0	0	0	1
Hsp-13408_c23	1	0	0	0	0	1	0	0	0	1
Hsp-14409_c23	1	0	0	0	0	1	0	0	0	1
Hsp-15410_c23	1	0	0	0	0	1	0	0	0	1

Table 7. Overview of constraints present in XCSP3 instances (continued)

Instance	AllDifferent	Cumulative	Inverse	InDomain	NegativeTable	NoOverlap	NoOverlap2D	Regular	ShortTable	Table
Hsp-aux-10405_c23	1	0	0	0	0	1	0	0	0	1
Hsp-aux-11406_c23	1	0	0	0	0	1	0	0	0	1
Hsp-aux-12407_c23	1	0	0	0	0	1	0	0	0	1
Hsp-aux-13408_c23	1	0	0	0	0	1	0	0	0	1
Hsp-aux-14409_c23	1	0	0	0	0	1	0	0	0	1
Hsp-aux-15410_c23	1	0	0	0	0	1	0	0	0	1
Hsp-table-10405_c23	1	0	0	0	0	1	0	0	0	11
KidneyExchange-4-041_c23	1	0	0	0	0	0	0	0	0	0
KidneyExchange-4-051_c23	1	0	0	0	0	0	0	0	0	0
KidneyExchange-4-061_c23	1	0	0	0	0	0	0	0	0	0
KidneyExchange-4-071_c23	1	0	0	0	0	0	0	0	0	0
KidneyExchange-4-081_c23	1	0	0	0	0	0	0	0	0	0
LitPuzzle-10_c24	0	0	0	100	0	0	0	0	0	0
LitPuzzle-15_c24	0	0	0	225	0	0	0	0	0	0
MaximumDensityOscillatingLife-5-2_c24	0	0	0	0	0	0	0	0	50	0
MaximumDensityOscillatingLife-6-2_c24	0	0	0	0	0	0	0	0	72	0
MisteryShopper-4-06-1-6-0_c24	54	0	4	0	0	0	0	0	0	184
MisteryShopper-4-06-2-6-0_c24	52	0	4	0	0	0	0	0	0	176
MisteryShopper-5-08-1-6-0_c24	88	0	5	0	0	0	0	0	0	390
MisteryShopper-5-08-2-6-0_c24	86	0	5	0	0	0	0	0	0	380
MisteryShopper-5-10-2-6-0_c24	106	0	5	0	0	0	0	0	0	480
MisteryShopper-5-10-3-6-0_c24	104	0	5	0	0	0	0	0	0	470
MisteryShopper-6-12-3-6-0_c24	150	0	6	0	0	0	0	0	0	828
NonTransitiveDice-08-08-3_c23	0	0	0	0	0	0	0	0	0	0
NonTransitiveDice-10-10-3_c23	0	0	0	0	0	0	0	0	0	0
ProgressiveParty-rally-red12-05_c23	0	0	0	0	0	0	0	0	0	0
ProgressiveParty-rally-red12-07_c23	0	0	0	0	0	0	0	0	0	0
RIP-25-0-j060-01-01_c23	0	4	0	0	0	0	0	0	0	0
RIP-25-0-j090-01-01_c23	0	4	0	0	0	0	0	0	0	0
RIP-25-0-j120-01-01_c23	0	4	0	0	0	0	0	0	0	0
RIP-25-2-j060-20-01_c23	0	4	0	0	0	0	0	0	0	0
RIP-25-2-j090-20-01_c23	0	4	0	0	0	0	0	0	0	0
RubiksCube-10-4-20-0_c24	20	0	0	0	0	0	0	0	1	2
RubiksCube-10-4-20-3_c24	20	0	0	0	0	0	0	0	1	2
RubiksCube-11-4-20-0_c24	20	0	0	0	0	0	0	0	1	2
RubiksCube-11-4-20-3_c24	20	0	0	0	0	0	0	0	1	2
RubiksCube-12-4-20-1_c24	20	0	0	0	0	0	0	0	1	2
SocialGolfers-06-06-07_c24	60	0	0	0	0	0	0	0	0	2
SocialGolfers-cnt-06-06-07_c24	60	0	0	0	0	0	0	0	0	2
Sonet-s2ring02_c23	0	0	0	0	0	0	0	0	15	0
StillLife-05-05_c24	0	0	0	0	12	0	0	0	0	25
StillLife-08-08_c24	0	0	0	0	24	0	0	0	0	64
StillLife-08-10_c24	0	0	0	0	28	0	0	0	0	80
StillLife-wastage-05-05_c24	0	0	0	0	20	0	0	0	0	25
Subisomorphism-g34-g36_c24	1	0	0	0	0	0	0	0	0	480

Table 8. Overview of constraints present in XCSP3 instances (continued)

Instance	AllDifferent	Cumulative	Inverse	InDomain	NegativeTable	NoOverlap	NoOverlap2D	Regular	ShortTable	Table
TestScheduling-t020m10r03-1_c24	0	1	0	2	0	3	0	0	0	0
TestScheduling-t020m10r05-1_c24	0	1	0	5	0	5	0	0	0	0
TestScheduling-t020m10r10-1_c24	0	1	0	2	0	4	0	0	0	0
TestScheduling-t030m10r03-1_c24	0	1	0	4	0	3	0	0	0	0
TestScheduling-t030m10r05-1_c24	0	1	0	7	0	5	0	0	0	0
TestScheduling-t030m10r10-1_c24	0	1	0	3	0	10	0	0	0	0
TestScheduling-t030m20r03-1_c24	0	1	0	8	0	3	0	0	0	0
TestScheduling-t030m20r05-1_c24	0	1	0	5	0	5	0	0	0	0
TestScheduling-t050m10r03-1_c24	0	1	0	10	0	3	0	0	0	0
TestScheduling-t050m10r05-1_c24	0	1	0	8	0	5	0	0	0	0
TestScheduling-t050m10r10-1_c24	0	1	0	11	0	10	0	0	0	0
TestScheduling-t050m20r03-1_c24	0	1	0	8	0	3	0	0	0	0
TestScheduling-t100m10r03-1_c24	0	1	0	17	0	5	0	0	0	0
TestScheduling-t100m20r03-1_c24	0	1	0	15	0	3	0	0	0	0
TestScheduling-t100m20r05-1_c24	0	1	0	23	0	5	0	0	0	0
TravelingTournament-galaxy04_c24	6	0	0	28	0	0	0	4	28	0
WordGolf-4-ogd2008-50-0_c24	0	0	0	0	0	0	0	0	0	2
WordGolf-4-ogd2008-50-3_c24	0	0	0	0	0	0	0	0	0	2