Solving complex problems using model transformations: from set constraint modeling to SAT instance solving

Frédéric Lardeux, Éric Monfroy, Eduardo Rodriguez-Tello, Broderick Crawford, Ricardo Soto

Solving complex problems using model transformations: from set constraint modeling to SAT instance solving

Frédéric Lardeux[a], Éric Monfroy[b], Eduardo Rodriguez-Tello[c,*], Broderick Crawford[d], Ricardo Soto[d]

[a]*LERIA, University of Angers. 2 Boulevard Lavoisier, 49045 Angers, France*
[b]*LS2N - UMR 6004. University of Nantes. 2 Rue de la Houssinière, 44322 Nantes, France*
[c]*Cinvestav Tamaulipas. Km. 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., Mexico*
[d]*Pontificia Universidad Católica de Valparaíso. Avenida Brasil 2950, Valparaiso 2362807, Chile*

Abstract

On the one hand, solvers for the propositional satisfiability problem (SAT) can deal with huge instances composed of millions of variables and clauses. On the other hand, Constraint Satisfaction Problems (CSP) can model problems as constraints over a set of variables with non-empty domains. They require combinatorial search methods as well as heuristics to be solved in a reasonable time.

In this article, we present a technique that benefits from both expressive CSP modeling and efficient SAT solving. We model problems as CSP set constraints. Then, a propagation algorithm reduces the domains of variables by removing values that cannot participate in any valid assignment. The reduced CSP set constraints are transformed into a set of suitable SAT instances. They may be simplified by a preprocessing method before applying a standard SAT solver for computing their solutions.

The practical usefulness of this technique is illustrated with two well-known problems: a) the Social Golfer, and b) the Sports Tournament Scheduling. We obtained competitive results either compared with *ad hoc* solvers or with hand-written SAT instances. Compared with direct SAT modeling, the proposed technique offers higher expressiveness, is less error-prone, and is relatively simpler to apply. The automatically generated propositional satisfiability instances are rather small in terms of clauses and variables. Hence, applying the constraint propagation phase, even huge instances of our problems can be tackled and efficiently solved.

Keywords: Model Transformations, Constraint Programming, Set Constraints, SAT Encoding, Combinatorial Problem

1. Introduction

Constraint Satisfaction Problems (CSP) can generally be used to easily model combinatorial problems (Rossi et al., 2006). A CSP consists of a set of variables (generally given with their domains or candidate values) and

*Corresponding author.
Email addresses: `Frederic.Lardeux@univ-angers.fr` (Frédéric Lardeux), `Eric.Monfroy@univ-nantes.fr` (Éric Monfroy), `ertello@tamps.cinvestav.mx` (Eduardo Rodriguez-Tello), `broderick.crawford@ucv.cl` (Broderick Crawford), `ricardo.soto@ucv.cl` (Ricardo Soto)

some constraints imposing conditions that these variables must satisfy. A solution is thus a particular assignment for those variables that does not violate any of the given constraints. One of the essential characteristics of CSP is their expressiveness. Indeed, variables of several types are allowed, *e.g.*, finite integer domains, intervals of real numbers, floating-point domains, Boolean, and sets. Moreover, constraints may have various syntax and semantics such as set constraints, non-linear polynomial, Boolean, equality of terms, and linear arithmetic. Also, a particular type of constraints, called global constraints, significantly increase solving efficiency as well as expressiveness. Indeed, they introduce some new relations between variables and some new powerful and specific reduction algorithms. Among the numerous global constraints, we can mention two of the most common ones: the *alldifferent* constraint that imposes assigning different values to variables of a list; and the *cardinality* constraint that links a set to its number of elements.

The propositional satisfiability problem (SAT), see (Garey & Johnson, 1979), represents an alternative for the formulation of combinatorial problems. Given a well-formed Boolean formula, usually expressed in Conjunctive Normal Form (CNF),[1] The SAT problem consists of finding a truth assignment for the literals that satisfies it. SAT expressiveness is, however, limited by its nature to propositional formulae and Boolean variables. It is thus a tedious task to directly code complex constraints into SAT, *e.g.*, see (Triska & Musliu, 2012), or (Gent & Lynce, 2005) for the coding of set constraints. Moreover, optimizing models concerning the number of variables and clauses could immediately produce very complicated models that are unreadable and more susceptible to include errors. Nowadays, the main strength of SAT solvers is the huge size of SAT instances (composed of millions of variables and clauses) that they can treat.

Thus, it is very interesting to design an efficient system for solving (engineering) problems modeled with sets constraints to:

- Encode CSPs into SAT formulas, such as in the work of Bessière et al. (2004) and Bacchus (2007), in order to take advantage of CSP expressiveness and modeling, as well as from the advanced capabilities of the state-of-the-art SAT solvers.
- Bring in a higher level of expressiveness into SAT modeling, *e.g.*, by introducing the use of some well-known global constraints like *alldifferent* (Lardeux et al., 2009), and *cardinality* (Bailleux & Boufkhad, 2003).

Compared to models based on matrices or integers, sets are convenient since they reduce the number of symmetries. Thus, it is well-known that numerous problems may be easily modeled with set constraints. Consequently, various constraint systems for dealing with set constraints appeared. Among them we find specialized systems and solvers (Azevedo, 2002, 2007), set specific libraries of constraint programming systems (Correas et al., 2018; Gervet, 1994), and set constraints integrated into the CHOCO solver (Prud'homme et al., 2017), to mention the most relevant of them.

[1]A CNF formula F is a conjunction of clauses C, each clause being a disjunction of literals, each literal being either a positive (x_i) or a negative ($\neg x_i$) propositional variable.

We are interested in various aspects of set constraints: CSP models with set constraints, reduction of finite domain and set variables, and the "encoding" of set constraints into SAT instances, which are finally solved with a standard SAT solver. The main objective of this work is not to outperform well-established CSP set solvers in terms of efficiency, but to ease the solving of set constraints with standard SAT solvers by using efficient model transformations. However, we obtained very good practical results in terms of efficiency. Indeed, for the Sports Tournament Scheduling problem, our generic technique is competitive with *ad hoc* solvers (see Section 8).

In a previous work (Lardeux et al., 2015), we proposed an efficient system, including a group of encoding rules that could be straightly applied to the CSP set constraint models. Nevertheless, some elements from set definitions could be eliminated without neither losing any solution nor changing the problem semantics. Hence, set variables domains could be made smaller, producing, in consequence, a narrower search space which translates into smaller SAT instances. Later, in Lardeux & Monfroy (2014), we proposed reductions for set variables as a constraint propagation process. This reduction was rather weak and could remove only some undesired elements from set variable domains. Although promising and fast, this reduction was not strong enough for drastically contracting the generated SAT instances. Thus, it has a reduced impact on SAT solving. Furthermore, the set representation that we had was not enough for modeling some relations, such as those for some symmetry breakings. In spite of these issues, the preliminary work presented in Lardeux & Monfroy (2016) showed that promising results could be obtained for the well-known Sports Scheduling problem.

In this article, we carry on building on our previous work presented in Lardeux et al. (2015) and Lardeux & Monfroy (2014) by proposing our new efficient system with the following new characteristics:

- **A new language for modeling CSP problems with set constraints (such as intersection, union, min, cardinality, . . .).** Although simple, our language contains every constraint provided in standard set solvers. Moreover, it is expressive and very intuitive. In comparison with the language previously introduced in Lardeux et al. (2015) and Lardeux & Monfroy (2014), we can claim that:
    - Our new language has been complemented with finite domain variables, *i.e.*, variables having finitely many candidate values such as bounded integer values.
    - Some comparison constraints between these new finite domain variables have been added, such as $A < B$.
    - The cardinality constraint connects now a variable having a finite domain with a set variable. In contrast, in Lardeux et al. (2015) and Lardeux & Monfroy (2014), it was only an attribute of a set that could not appear directly within other constraints.
    - Correas et al. (2018) proposed the addition of new types of set constraints as well as a more productive cooperation between finite domain variables and set variables. Similarly, we have also introduced constraints for connecting the minimum and maximum elements of a set with the set itself. These minimum

and maximum are also finite domain variables that other constraints can use. We found them of great utility for breaking symmetries (*e.g.*, for ordering several sets) and reducing sets.

- **A new set of stronger rules ($\Rightarrow_{red}$) for reducing CSP models during the propagation process:**
  - Every rule in this set corresponds to a reduction function (Apt, 2003) for finite domain variables and sets. Thus, the fixed point application of these rules defines a propagation algorithm.
  - The reduction depends on the computing of lower and upper bounds for the analyzed sets as well as their minimum and maximum cardinalities (whereas that presented in Lardeux & Monfroy (2016) employed only upper bounds). This reduction is similar to the one introduced by Azevedo (2002).
  - The new proposed reduction scheme is much stronger than that previously presented in Lardeux & Monfroy (2016). It is stronger than bound consistency for set constraints (*e.g.*, Gervet, 1994), and weaker than the one presented by Yip & Van Hentenryck (2011). This choice is discussed and justified in Section 9.
  - Last but not least, we have also added some rules for manipulating disjunctions of constraints. We are thus able to remove some tautologies (*i.e.*, some complete disjunctions may vanish), and some contradictions (*i.e.*, some disjuncts of a disjunction may vanish). To our knowledge, this had never been proposed in a CSP set solver.
- **Some more efficient encoding rules ($\Leftrightarrow_{enc}$) for translating CSP models into propositional satisfiability instances:**
  - Additional rules complete those presented in (Lardeux & Monfroy, 2014). They permit to convert the new set constraints implemented in this paper.
  - These additional rules apply to constraints without or after propagation, and without generating useless clauses (as it was sometimes the case in our previous work). Succinctly, the process is the following for each type of constraint: for every set participating in a given constraint and for every element of the universe, we consider three mutually exclusive membership cases:
    - The element is in the lower bound of the set, *i.e.*, the element is effectively in the set.
    - The element is not in the lower bound of the set, but it is in the upper bound, *i.e.*, the set could eventually contain the element.
    - The element is not in the upper bound of the set, *i.e.*, the set definitively does not contain this element.

    Hence, for a single ternary constraint (*e.g.*, $A = B \cap C$), 27 cases should be considered. After applying our new reduction rules ($\Rightarrow_{red}$), some of these cases may never be fulfilled. Hence, our new encoding rules do not generate any clauses in these cases, and consequently, the generated SAT instances are smaller.

We applied our technique successfully to diverse classic problems, including Sudoku, *n*-queens, car sequencing, and WhoWithWhom. To illustrate this paper, we have decided to use two challenging problems: the Social Golfer

Problem (SGP) (Harvey, 2019) and the Sports Tournament Scheduling problem (STS) (Walsh, 2019). The complexity of our automatically generated propositional satisfiability instances is similar to that of other improved and handwritten SAT instances. Moreover, our SAT instances appear to be much smaller than the ones we used to generate before. Thus, larger problems can be tackled, even problems which used to cause memory overflow problems. Furthermore, solving our automatically generated SAT instances by using a standard propositional satisfiability solver (*e.g.*, *MiniSAT* reported in Eén & Sörensson (2004)) could produce very competitive results compared to other SAT solving methods. For instance, our approach can match the global performance of the algorithm reported in Hamiez & Hao (2014) for the STS problem, which is, as far as we know, the most efficient algorithm for the STS problem. This algorithm was specially devised for solving the STS problem. However, it may produce unsatisfiable instances given its over-constrained problem definition.

The following section (Section 2) gives an overview of our method and some motivations. Section 3 presents the notion of set CSP and our set constraint language. In Section 4, we present some of our reduction rules, over finite domain variables, sets, and disjunctions. In Section 5, we then discuss some implementation issues to obtain a more efficient constraint propagation process. The rules for encoding CSP instances into SAT instances are described in Section 6. Section 7 illustrates the use of set constraints for modeling the STS and SGP problems. These two problems are also used in Section 8 to evaluate our approach by considering the efficiency of reduction rules and SAT preprocessing. We analyze the methods in the literature and their limits, and then we show how our method can overcome some of these problems in Section 9. We finally conclude in Section 10.

2. Overview of the approach

Our main goal is to provide expressive techniques for generating propositional satisfiability instances, which can be then solved by an existing SAT solver. To this end, we work at the level of models, instances, and model and instance transformations and conversions. In this article, our approach is made up of the following steps (Figure 1 gives a simplified view):

1. **A problem is modeled with CSP set constraints: this gives a CSP model.** The CSP model can be generated by a modeler, such as Savile Row (Nightingale & Miguel, 2018) or *MiniZinc* (Nethercote et al., 2007; Stuckey et al., 2014). However, our language evolves continuously. For example, we are currently experimenting with some specific partitioning constraints and patterns of conjunctions of constraints which have special reduction and encoding rules. Thus, we preferred to use standard languages to keep flexibility, either C++ or Prolog (SWI-Prolog proposed by Wielemaker et al., 2012). The CSP models are thus Prolog or C++ programs that formulate the variables and the constraints.

   We consider usual sets constraints (such as $\cup$ and $\cap$), constraints enforcing minimum and maximum values of sets (such as in Correas et al. (2018)), and minimum and maximum cardinality of sets. These minimum and

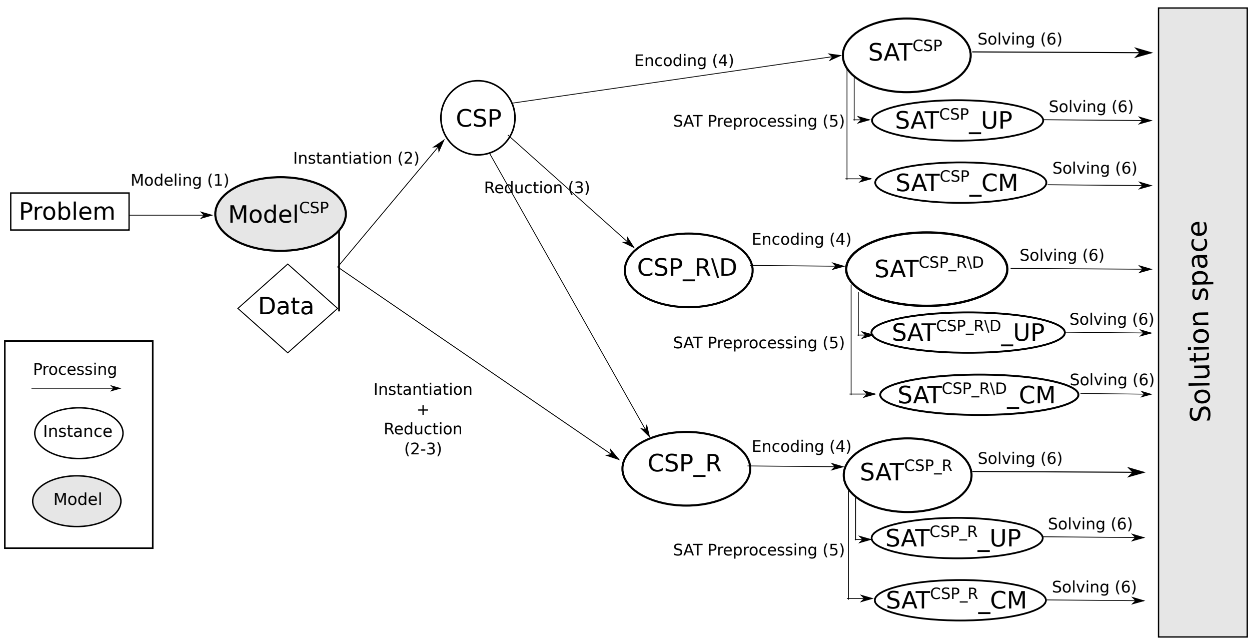

Figure 1: Models, instances, and their processing.

maximum values and cardinality are finite domain variables. These constraints did not exist in our previous work.

2. **A CSP model, together with some data, leads to a CSP instance.** For example, let us consider we have the Social Golfer model. Then, this CSP model (Model$^{\text{CSP}}$) and the number of players set to 5, the number of groups set to 3, and the number of weeks set to 4 leads to the CSP instance 5-3-4 of the Social Golfer.

   Since XCSP3 (Boussemart et al., 2017) was not released when we started this work, we designed our XML-like format for CSP instances. Thus, the Prolog and C++ programs that represent CSP models (and data) generate the CSP instances in our XML-like format.

3. **A CSP instance is reduced by a propagation process in order to get a reduced CSP instance.** The propagation process computes a fixed point of our reduction rules, leading to a reduced CSP instance. CSP_R is the CSP instance reduced by all the rules. CSP models can also lead directly to CSP_R instances using dynamic reduction (see Section 5.4 for details). Reduction without rules for disjunction constraints leads to CSP_R\D instances. For finite domain variables, we use arc-consistency like in Mackworth (1992). A reduction rule tries to push up lower bounds and to push down upper bounds of set variables using the current bounds, and the minimum and maximum cardinalities. This consistency for sets has been defined in Azevedo (2002) (see Section 4 for details). We complete this reduction by removing constraints that become tautologies: indeed, these useless constraints can generate extra variables and clauses in the SAT instances.

Another novelty is also to treat the disjunctions of constraints (CSP_R). This reduction consists in detecting disjunctions that are tautologies to remove them. Disjuncts that are contradictions are also detected and removed. This simplification can drastically reduce some instances. More details are given in Section 4.4. Note that our new reduction is much stronger than the one of Lardeux & Monfroy (2014) which only treated upper bounds of sets.

4. **A CSP instance (reduced or not) is encoded into a SAT instance.** Our new encoding is more efficient than the one of Lardeux et al. (2015). Indeed, we can now consider either reduced or non-reduced CSP instances as input, without generating useless SAT variables and clauses (see Section 6). We then respectively obtain reduced SAT models ($\mathsf{SAT}^{\mathsf{CSP\_R}}$) or SAT models ($\mathsf{SAT}^{\mathsf{CSP}}$).

5. **SAT instances ($\mathsf{SAT}^{\mathsf{CSP}}$) can be processed before being sent to the SAT solver.** For example, one can perform unit propagation or some steps of resolution. In this article, we used the *SatELite* (Eén & Biere, 2005) preprocessor. We report about its effects on resolution in Section 8.2. Note that $\mathsf{SAT}^{\mathsf{CSP}}$_UP, $\mathsf{SAT}^{\mathsf{CSP}}$_CM, and $\mathsf{SAT}^{\mathsf{CSP\_R}}$ are not necessarily the same. Reduction is processed before the SAT encoding for $\mathsf{SAT}^{\mathsf{CSP\_R}}$ and after for $\mathsf{SAT}^{\mathsf{CSP}}$_UP and $\mathsf{SAT}^{\mathsf{CSP}}$_CM. All SAT instances suffixed by _UP correspond to SAT instances only preprocessed by unit propagation. Similarly, all SAT instances suffixed by _CM correspond to SAT instances preprocessed by a complete CNF minimizer (see Section 8.2 for details).

6. **A SAT instance is solved with a SAT solver.** In our case, we use *MiniSAT* because it is a standard, classical, and complete solver (*i.e.,* if there is a solution, it finds the solution; otherwise, it proves the unsatisfiability of the instance). None of the processings modifies the solution space of the problem. However, we are interested in the first solution returned by the solving. Thus, each branch can lead to a different solution.

Some of these steps are not mandatory. We see later that we can skip Step 5: preprocessing SAT instances is not always beneficial (Subsection 8.2.2). Step 3 is optional: indeed, the approach presented in Lardeux et al. (2015) corresponds to Steps 1-2-4-6. However, we show in this paper that the reduction step is crucial, in terms of the size of produced SAT instances, and also in terms of the total solving time (Step 6). The method in Lardeux & Monfroy (2014) corresponds to 1-2-3-4-6, but with a much weaker reduction than the one we are using now. We also show that Steps 2 and 3 can be achieved at the same time (see Section 5.4): in this case, CSP instances are reduced sooner, and some larger problems can be tackled and reduced. However, in this case, we do not have the raw initial CSP instance (but this is also the case with *MiniZinc* (Nethercote et al., 2007; Stuckey et al., 2014) which generates a *FlatZinc* instance, which is already transformed, for example, by some reductions, and modifications of variables.)

The advantages of our approach are:

- Problems are expressively modeled as CSPs, and we have a better expressiveness than before (Lardeux & Monfroy, 2014; Lardeux et al., 2015). We have added disjunctions, finite domain variables, minimum and maximum cardinalities, and minimum and maximum of sets as variables. We now have the expressiveness of the set constraint language of Cardinal (Azevedo, 2007) plus disjunction. Symmetry breaking techniques can be easily added as extra constraints.

- This technique is less error-prone than hand-written propositional satisfiability instances. For instance, in (Triska & Musliu, 2012) the authors had to revise the model for the Social Golfer Problem presented in (Gent & Lynce, 2005), since they found some corrections to the ranges of different disjunctions ($\vee$) and conjunctions ($\wedge$) composing the clauses of the hand-written SAT instances.
- The generated SAT instances are even more compact than those presented in Lardeux et al. (2015). In the particular case of the SGP, the SAT instances generated with our new technique are smaller (in terms of clauses) than the instances presented in Triska & Musliu (2012).
- Our generated SGP instances are more appropriate for SAT solvers, as they can be solved faster than those reported in (Triska & Musliu, 2012). Concerning the STS problem (Section 7.2), the proposed approach attains a performance that successfully compares with that of the best-known *ad hoc* solver (Hamiez & Hao, 2014), which has the disadvantage of producing an over-constrained model that leads to a loss of solutions (unsatisfiable instances).
- The reduction process (which is stronger than before Lardeux & Monfroy, 2014) is an overhead which is generally compensated by faster solving times. When instances contain a huge number of disjunctions, the reduction is efficient in terms of SAT instance size but may become inefficient in terms of expended CPU time (see Section 8).
- The model is reduced before generating SAT instances, and the encoding process is more efficient than before (it does not generate useless clauses or variables). Hence, generated SAT instances are consequently smaller, and SAT solvers can now handle larger problems (that could not be treated before because of their size).

To show the difference with our previous work, Figure 2 illustrates the processes presented in Lardeux et al. (2015).

3. CSP set constraints

In this section, we define the set constraint language that we consider in the remaining sections: notion and declaration of variables, universe, and set constraints.

3.1. Set-CSP

Definition 1 (Set-CSP). *A Set-CSP is a four-tuple* $(\mathcal{U}, X, \mathbb{F}, C)$ *such that:*

- $\mathcal{U}$ *denotes the universe, i.e., a finite set of integers.*

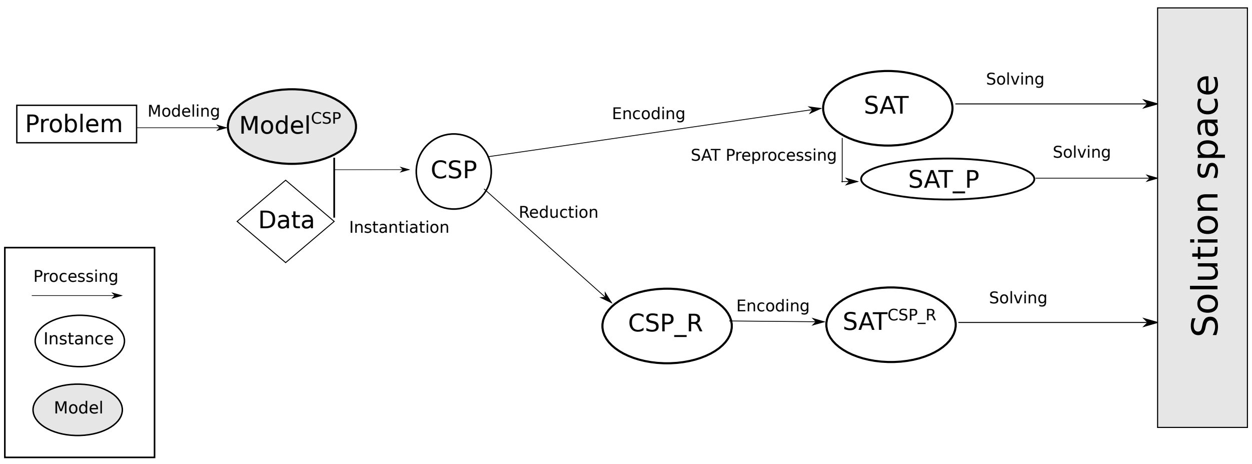

Figure 2: Models, instances, and processing presented in Lardeux et al. (2015).

- *$X = \{x_1, \ldots, x_n\}$ is a set of variables, where each variable $x_i \in X$ has a finite domain $D_{x_i} \subseteq \mathcal{U}$. The Cartesian product $D_{x_1} \times \ldots \times D_{x_n}$ is represented as $\mathcal{D}$.*
- *$\mathbb{F}$ is a set of set variables, and for each $F \in \mathbb{F}$:*
  - *$\underset{\cdot}{F} \subseteq \mathcal{U}$, which denotes the greatest lower bound of F, is a set that contains elements of the universe that are necessarily in F.*
  - *$\dot{F} \subseteq \mathcal{U}$, which denotes the lowest upper bound of F, is a set that contains elements of the universe that may be in F.*
  - *Also, $\lfloor F \rfloor \in \mathbb{N}$, $\lceil F \rceil \in \mathbb{N}$ represent the minimum and maximum cardinality of F, respectively.*

  Given a set F with cardinality $|F|$, then the following conditions are fulfilled: $\underset{\cdot}{F} \subseteq F \subseteq \dot{F}$, $\lfloor F \rfloor \leq |F| \leq \lceil F \rceil$, $|\underset{\cdot}{F}| \leq \lfloor F \rfloor$, and $\lceil F \rceil \leq |\dot{F}|$.
- *C is used to express a set of constraints relating variables defined over the Cartesian product $\mathcal{D}^{|X|} \times \mathcal{U}^{|\mathbb{F}|}$.*

$\dot{F} \setminus F^? = \underset{\cdot}{F}$ represents those elements required in F, while $F^? = \dot{F} \setminus \underset{\cdot}{F}$ stands for the elements that could be eventually in F. Hence, $\underset{\cdot}{F} \subseteq F \subseteq \dot{F}$, as well as the solutions for F, belong to the powerset $2^{\dot{F}}$. Moreover, they include all the elements in the greatest lower bound (that also always pertain to the least upper bound), and their cardinality is an integer in the interval $[\lfloor F \rfloor..\lceil F \rceil]$.

3.2. Elementary set constraints

In this work, the constants and the variables are defined as follows:

- Let $\mathcal{U} :: \top$ be the universe, with $\top$ representing a set.

- Let $x :: D_x$ defines a finite domain variable x with domain D_x being a set of elements.
- Let $F :: (\underset{\cdot}{F}, \dot{F}, \lfloor F \rfloor, \lceil F \rceil)$ be a set variable, with the sets of elements $\underset{\cdot}{F}$, $\dot{F}$ denoting its lower and upper bounds, and the integer values $\lfloor F \rfloor$, $\lceil F \rceil$ representing its minimum and maximum cardinalities.
- Let $\emptyset :: (\emptyset, \emptyset, 0, 0)$ represent a set variable having no elements (empty).

Assume that F, G, H, and F_i ($i \in [1..n]$) are set variables. Assume also that x is a finite domain variable. Our modeling language contains the following usual set constraints:

finite domain (dis)equality	$x = y$	$(x \neq y)$
finite domain (strict) inequality	$x \leq y$	$(x < y)$
(non)membership	$x \in F$	$(x \notin F)$
set (dis)equality	$F = G$	$(F \neq G)$
inclusion	$F \subseteq G$	$(F \nsubseteq G)$
difference	$H = F \setminus G$	
intersection	$F = \bigcap_{i=1}^{n} F_i$	
union	$F = \bigcup_{i=1}^{n} F_i$	
partition	$F = \bigsqcup_{i=1}^{n} F_i$	
set cardinality	$x = \lvert F \rvert$	
set variable minimum	$x = min(F)$	
set variable maximum	$x = max(F)$	

Note that $F = \bigsqcup_{i=1}^{n} F_i$ is equivalent to $F = \bigcup_{i=1}^{n} F_i$, and $F_i \cap F_j = \emptyset$ for all $i \neq j$.

Constraints are linked by conjunctions and disjunctions (and thus implication). We also include quantification ($\exists$, and $\forall$) over closed sets, which are syntactic sugar (respectively disjunctions and conjunctions).

Example 1. $\forall i \in [1..n]\ G_i \subseteq G_{i+1}$ *enforces a chain of included sets* G_j.
$G :: (\{1,2,3,4\}, \{1,2,3,4\}, 4, 4) \ \wedge\ J :: (\emptyset, \{1,2,3,4,5,6,7,8\}, 3, 3) \ \wedge\ \forall p \in G,\ p \notin J$ *creates a closed set* $G = \{1,2,3,4\}$*, a set J of cardinality 3 which contains three elements among* $1,2,3,4,5,6,7,8$. *Then, elements of G are forced not to be in J.*

4. Reduction rules

Rules are an elegant way to describe and formalize changes to be applied to an object (either a simple object or a complex structure). Furthermore, this formalism is rather classic for expert systems and constraint transformations. See, for example, Frühwirth (2009); Ftulis et al. (1998); Lee & Kwon (1995); Topaloglu et al. (2012). That is the reason why we formalize our reduction algorithm as the fixed point of a set of transformation rules.

The goal of the reduction rules ($\Rightarrow_{red}$) is to bring down the size of the CSP search spaces. A constraint propagation method for reducing sets, as well as finite domains, is defined by a fixed point application of these rules. Every reduction rule may:

- Add (respectively eliminate) some elements to lower bounds (respectively from upper bounds) of set variables.
- Increase (respectively decrease) minimum cardinalities (respectively maximum cardinalities) of set variables.
- Eliminate integers from the domains of finite domain variables.
- Lead to failure cases when there is no solution (*e.g.*, when a domain is empty, or a minimum cardinality is bigger than a maximum cardinality).
- Also, remove some constraints that became tautologies, and thus useless.

Note that it is not mandatory to have reduction rules for each constraint: their role is only to simplify the encoding and, thus, the work of the SAT solver. However, we have rules for each of the constraints presented above. The fixed point application of our reduction rules enforces bound consistency with cardinality (Azevedo, 2002, 2007). In the next subsections, we present a selection of our reduction rules. Classical logical operators as $\leftarrow$ or $\models$ are also used to define $\Rightarrow_{red}$.

4.1. Finite domains

When the domain of a variable x is empty, there is no solution for the CSP:

$$D_x = \emptyset \quad \Rightarrow_{red} \quad fail \tag{1}$$

If we declare twice a variable x, both declarations are contracted into a single one:

$$x :: D_x,\;\; x :: D'_x \quad \equiv_{red} \quad x :: D_x \cap D'_x \tag{2}$$

4.2. Sets

Some rules may add some elements to the set $\underset{\cdot}{F}$ that may not belong to $\dot{F}$. Thus, Rule (4) is essential because it leads to a failure in these circumstances. Rules (3), (5), and (6) are similar: they are beneficial when modifications are made to the lower or upper bound of a set cardinality.

$$\lfloor F \rfloor > \lceil F \rceil \quad \Rightarrow_{red} \quad fail \tag{3}$$

$$\underset{\cdot}{F} \not\subseteq \dot{F} \quad \Rightarrow_{red} \quad fail \tag{4}$$

$$|\underset{\cdot}{F}| > \lceil F \rceil \quad \Rightarrow_{red} \quad fail \tag{5}$$

$$|\dot{F}| < \lfloor F \rfloor \quad \Rightarrow_{red} \quad fail \tag{6}$$

If $\lceil F \rceil = 0$, then F corresponds to a set having no elements (moreover, if $\underset{\cdot}{F} \neq \emptyset$, then applying Rule (4) results in a failure):

$$\lceil F \rceil = 0 \quad \Rightarrow_{red} \quad \dot{F} = \emptyset \tag{7}$$

Rule (8) fixes the set F when the upper bound cardinality equals its minimum cardinality. Respectively, Rule (9) fixes the set F when the lower bound cardinality equals its maximum cardinality.

$$\lfloor F \rfloor = |\dot{F}|,\ \ \underset{\cdot}{F} \subset \dot{F},\ \ \lfloor F \rfloor \leq \lceil F \rceil \quad \Rightarrow_{red} \quad \underset{\cdot}{F} \leftarrow \dot{F},\ \ \lceil F \rceil \leftarrow \lfloor F \rfloor \tag{8}$$

$$\lceil F \rceil = |\underset{\cdot}{F}|,\ \ \underset{\cdot}{F} \subset \dot{F},\ \ \lfloor F \rfloor \leq \lceil F \rceil \quad \Rightarrow_{red} \quad \dot{F} \leftarrow \underset{\cdot}{F},\ \ \lfloor F \rfloor \leftarrow \lceil F \rceil \tag{9}$$

Moreover, Rules (10) and (11) can only be triggered once to match the different fields of a set declaration:

$$|\underset{\cdot}{F}| > \lfloor F \rfloor \quad \Rightarrow_{red} \quad \lfloor F \rfloor \leftarrow |\underset{\cdot}{F}| \tag{10}$$

$$|\dot{F}| < \lceil F \rceil \quad \Rightarrow_{red} \quad \lceil F \rceil \leftarrow |\dot{F}| \tag{11}$$

4.3. Set constraints

Rule (12) allows us to reduce three different set variables, F, G, and H, which are related by a set difference constraint. It is important to remark that in the right-hand side of a rule, multiple assignments ($\leftarrow$) are achieved at the same time. We mean that sets on the right-hand side (respectively left-hand side) of an assignment are the values of the set before (respectively after) applying the rule. Thus, if a set S appears several times on the right hand-side of some assignments, it always has the same value (*i.e.*, its value before applying the rule). Note also that $min\{a_1, \ldots, a_n\}$ (respectively $max\{a_1, \ldots, a_n\}$) returns the smallest (respectively the biggest) integer a_i.

$$H = F \setminus G \quad \Rightarrow_{red} \quad \begin{cases} \begin{array}{lcl} \dot{H} & \leftarrow & (\dot{H} \cap \dot{F}) \setminus \underset{\cdot}{G} \\ \dot{F} & \leftarrow & \dot{F} \cap (\dot{H} \cup \dot{G}) \\ \dot{G} & \leftarrow & \dot{G} \setminus \underset{\cdot}{H} \\ \underset{\cdot}{H} & \leftarrow & \underset{\cdot}{H} \cup (\underset{\cdot}{F} \setminus \dot{G}) \\ \underset{\cdot}{F} & \leftarrow & \underset{\cdot}{H} \cup \underset{\cdot}{F} \\ \underset{\cdot}{G} & \leftarrow & \underset{\cdot}{G} \\ \lfloor H \rfloor & \leftarrow & \max\{\lfloor H \rfloor,\ |\underset{\cdot}{H} \cup (\underset{\cdot}{F} \setminus \dot{G})|\} \\ \lfloor F \rfloor & \leftarrow & \max\{\lfloor F \rfloor,\ |\underset{\cdot}{H} \cup \underset{\cdot}{F}|\} \\ \lfloor G \rfloor & \leftarrow & \lfloor G \rfloor \\ \lceil H \rceil & \leftarrow & \min\{\lceil H \rceil,\ |(\dot{H} \cap \dot{F}) \setminus \underset{\cdot}{G}|\} \\ \lceil F \rceil & \leftarrow & \min\{\lceil F \rceil,\ |\dot{F} \cap (\dot{H} \cup \dot{G})|\} \\ \lceil G \rceil & \leftarrow & \min\{\lceil G \rceil,\ |\dot{G} \setminus \underset{\cdot}{H}|\} \end{array} \end{cases} \tag{12}$$

Once Rule (12) has been applied, if $F^{?} \cap \dot{G} = \emptyset$ then $H = F \setminus G$ always evaluates to true. Hence, this constraint is useless and can vanish. Such conditions are evaluated for every class of constraints: they can scale down the CSP instance size, and in consequence, the size of the generated propositional satisfiability instance.

We have also integrated some redundant rules that do not impact the reduction strength of our method. These rules are specializations of some other generic rules, and they apply faster. For example, the generic rule for the constraint $H = F \cap G$ is:

$$H = F \cap G \quad \Rightarrow_{red} \quad \left\{ \begin{array}{lll} \dot{H} & \leftarrow & \dot{H} \cap \dot{F} \cap \dot{G} \\ \underset{\cdot}{H} & \leftarrow & \underset{\cdot}{H} \cup (\underset{\cdot}{F} \cap \underset{\cdot}{G}) \\ \underset{\cdot}{F} & \leftarrow & \underset{\cdot}{F} \cup \underset{\cdot}{H} \\ \underset{\cdot}{G} & \leftarrow & \underset{\cdot}{G} \cup \underset{\cdot}{H} \\ \dot{F} & \leftarrow & \dot{F} \setminus (\underset{\cdot}{G} \setminus \dot{H}) \\ \dot{G} & \leftarrow & \dot{G} \setminus (\underset{\cdot}{F} \setminus \dot{H}) \\ \lfloor F \rfloor & \leftarrow & max\{\lfloor F \rfloor, |\underset{\cdot}{F} \cup \underset{\cdot}{H}|\} \\ \lfloor G \rfloor & \leftarrow & max\{\lfloor G \rfloor, |\underset{\cdot}{G} \cup \underset{\cdot}{H}|\} \\ \lfloor H \rfloor & \leftarrow & max\{\lfloor H \rfloor, \ |\underset{\cdot}{H} \cup (\underset{\cdot}{F} \cap \underset{\cdot}{G})|\} \\ \lceil F \rceil & \leftarrow & min\{\lceil F \rceil, |\dot{F} \setminus (\underset{\cdot}{G} \setminus \dot{H})|\} \\ \lceil G \rceil & \leftarrow & min\{\lceil G \rceil, |\dot{G} \setminus (\underset{\cdot}{F} \setminus \dot{H})|\} \\ \lceil H \rceil & \leftarrow & min\{\lceil H \rceil, \lceil F \rceil, \lceil G \rceil, |\dot{H} \cap \dot{F} \cap \dot{G}|\} \end{array} \right. \tag{13}$$

When $H = \emptyset$, the rule is much simpler but still equivalent. Practically, less computation steps and less tests are needed:

$$\emptyset = F \cap G \quad \Rightarrow_{red} \quad \left\{ \begin{array}{lll} \dot{F} & \leftarrow & \dot{F} \setminus \underset{\cdot}{G} \\ \dot{G} & \leftarrow & \dot{G} \setminus \underset{\cdot}{F} \\ \lceil F \rceil & \leftarrow & \min\{\lceil F \rceil, |\dot{F} \setminus \underset{\cdot}{G}|\} \\ \lceil G \rceil & \leftarrow & \min\{\lceil G \rceil, |\dot{G} \setminus \underset{\cdot}{F}|\} \end{array} \right. \tag{14}$$

4.4. Disjunctions

Disjunctions may be practical for modeling problems since they introduce even more expressivity. However, their treatment in terms of propagation quickly becomes tedious.

Consider a CSP given as $V \wedge C \wedge (d_1 \vee \ldots \vee d_n)$ where V represents variable declarations (set and finite domain variables), C and the d_i's are formulas built with conjunctions and disjunctions of set constraints. We consider that each d_i does not contain any variable declaration (this is not a restriction: declarations in each d_i can raise to V). Propagation from $V \wedge C$ to $(d_1 \vee \ldots \vee d_n)$ thus only consists of removing constraints. Propagation from $d_1 \vee \ldots \vee d_n$

to $V \wedge C$ is not correct in the general case. Consider $d_1 \vee \ldots \vee d_n \equiv d \wedge (d'_1 \vee \ldots \vee d'_n)$. Then, propagation from d to $V \wedge C$ can apply. However, we consider that in such a case, d can raise to C.

We are thus concerned here with propagation from $V \wedge C$ to $(d_1 \vee \ldots \vee d_n)$ to remove constraints, either complete disjunctions or just some disjuncts:

1. If $V \wedge C \models d_i$, then $d_1 \vee \ldots \vee d_n$ can be replaced by *true*
2. If $V \wedge C \models \neg d_i$, then $d_1 \vee \ldots \vee d_n$ can be simplified into $d_1 \vee \ldots \vee d_{i-1} \vee d_{i+1} \vee \ldots \vee d_n$

The first case always works, while the second case is tractable when we have the negation of the constraint (*e.g.*, $=$ and its negation $\neq$). In our case, only set difference and disjoint union constraints do not have direct negation.

Each type of constraint d_i requires its proper rules for testing $V \wedge C \models d_i$ or $V \wedge C \models \neg d_i$ and treating disjunction. In the following, we give disjunction reduction rules for the non-membership constraint ($\notin$).

The first rule enables us to remove a disjunct (which is a $\notin$ constraint) from a disjunction. The c_i's are conjunctions and disjunctions of set constraints. If the domain of x is included in the lower bound of G, then, $x \notin G$ is always false and it can be removed from the disjunction:

$$\begin{gathered} c_1 \vee \ldots \vee c_{i-1} \vee x \notin G \vee c_{i+1} \vee \ldots \vee c_n, \quad D_x \subseteq \underset{\cdot}{G} \\ \Rightarrow_{red} \\ c_1 \vee \ldots \vee c_{i-1} \vee c_{i+1} \vee \ldots \vee c_n \end{gathered} \tag{15}$$

The next rule replaces a disjunction by *true*. When the domain of x does not have any value in common with the upper bound of the set G, then the constraint $x \notin G$ is always true, and consequently, the disjunction $c_1 \vee \ldots \vee c_n$ can be replaced by *true*:

$$\begin{gathered} c_1 \vee \ldots \vee c_{i-1} \vee x \notin G \vee c_{i+1} \vee \ldots \vee c_n, \quad D_x \cap \dot{G} = \emptyset \\ \Rightarrow_{red} \\ true \end{gathered} \tag{16}$$

5. Some implementation considerations for the reduction rules

In this section, we present some aspects of the implementation of the reduction rules. The whole software is composed of 3 separate parts:

1. The modelization module, which is written in SWI-Prolog (Wielemaker et al., 2012).
2. The reduction module, which is written in CHR (Frühwirth, 2009). Note that we use the SWI-Prolog version of CHR.

3. The encoding module, which is written in C++.

The reduction module thus consists of CHR rules (corresponding to our $\Rightarrow_{red}$ rules) and some Prolog predicates. We present here 4 aspects of the reduction module:

1. The input syntax.

2. The structure we used for domains.

3. The splitting of $\Rightarrow_{red}$ rules into several CHR rules for efficiency reasons.

4. The dynamic reduction to apply $\Rightarrow_{red}$ rules as soon as a new constraint of the model is formulated; this overcomes some memory problems.

5.1. Syntax

Without loss of expressivity, we restrict CSP instances to be of the form $V \wedge C$ where D is a conjunction of set, and finite domain variable declarations and C is a conjunction of basic constraints or disjunctions of basic constraints, *i.e.*, $C = c_1 \wedge \ldots \wedge c_n$ where c_i is either a basic constraint or a disjunction of basic constraints $c_i = \vee_j c_{i,j}$. If this syntax is not respected, this is not a problem, and the encoding still works. However, propagation will be less efficient and will treat only a part of the CSP (constraints respecting this format). For efficiency reasons, it is thus advised to put disjunctions at the end of the model (once again, this is not mandatory). To simplify modeling, we plan to automate constraint re-ordering in the future.

5.2. Domains

To represent domains of finite domain variables and domains of set variables, we need to represent sets. A representation in extension of sets (*e.g.*, with some Prolog lists) is enough. However, our reduction rules require applying numerous set operations (*e.g.*, $\cup$ or $\cap$) on these domains either to perform some tests before triggering some rules or to achieve computation inside the rule (*e.g.*, new lower and upper bounds, and new variable domain). These operations are very costly for sets given in extension. Thus, we use the concept of s_interval for lower and upper bounds of sets and domains of variables. Set operations (*e.g.*, $\cup$ or $\cap$) are more efficient over this structure:

- An interval I of integers is denoted by $n..m$ where n and m are integers; $I = n..m$ represents every integer between n and m; the lower bound of I is denoted $\underline{I}$ and the upper bound $\overline{I}$.

- An s_interval is an ordered sequence of disjoint intervals:[2]

$$Li = (L_1, \ldots, L_l)$$

[2]Consider $I = a..b$, and $J = c..d$. Then, I and J are disjoint iff $b < c$ or $d < a$, and $I < J$ iff $b < c$.

The empty s_interval is denoted by $\perp$. An s_interval $I = (I_1, \ldots, I_n)$ is included in an s_interval $J = (J_1, \ldots, J_m)$ if each integer appearing in I also appears in J:

$$I \subseteq J \;\Leftrightarrow\; \forall I_i, \forall v \in [\underline{I_i}..\overline{I_i}], \exists J_k, \underline{J_k} \leq v \leq \overline{J_k}$$

The cardinality $|I|$ of an s_interval $I = (I_1, \ldots, I_n)$ is given by: $|I| = \sum_{i=1}^{n}(\overline{I_i} - \underline{I_i} + 1)$. The minimum $min(I)$ (respectively maximum) of an s_interval $I = (I_1, \ldots, I_n)$ is $min(I) = \underline{I_1}$ (respectively $max(I) = \overline{I_n}$). Other operations (such as $\cup$, $\cap$, …) on s_intervals are defined similarly. Note that these operations can be implemented simply by reasoning on bounds of intervals.

5.3. Splitting rules

Constraint Handling Rules (CHR) (Frühwirth, 2009) have been employed to implement the $\Rightarrow_{red}$ rules. CHR is a declarative, rule-based language whose concrete syntax depends on the host language, in our case SWI-Prolog. A CHR program, then, consists of rules that manipulate a multi-set of terms, called the *constraint store*: terms may be added or removed from the store. In our implementation, terms either represent variables (set or finite domain variables) together with their domains or constraints over these variables. Reducing the domain of a variable thus consists in removing the previous declaration of the variable, and adding a new declaration containing the reduced domain.

When added to the store, a constraint is active. This means that each rule where it matches the left-hand side is tried again. For efficiency reasons, it is thus crucial to change (remove, and then add) variable declaration only when the change is effective (otherwise, the system spends much time doing the same useless work again). Most of our reduction rules eventually reduce several sets or finite domain variables at once. For example, Rule (13) for the constraint $H = F \cap G$ may reduce H, F, and G. Such rules are split into several rules, one for each set that it eventually reduces. For example, Rule (13) is implemented as 3 rules (one for reducing H, one for reducing F, and one for G). Moreover, some guards verify that the rule effectively reduces the set or finite domain variables before being triggered. Thus, rules are applied only when they effectively reduce a (set or finite domain) variable. The CHR rule for reducing the set H with an intersection constraint $H = F \cap G$ is as follows:

```
set_intersection(H,F,G),
set(F,support(Fs,FS),card(Fc,FC)),
set(G,support(Gs,GS),card(Gc,GC)) \
set(H,support(Hs,HS),card(Hc,HC))  <=>
     % NHS is the new upper bound for H
     ord_int_intersection([FS,GS,HS],NHS),
     % NHs is the new lower bound for H
     ord_int_intersection(Fs,Gs,FGs),
     ord_int_union(Hs,FGs,NHs),
10      % NHC is the new max card for H
11      ord_int_length(NHS,SNHS),
12      min_list([HC,FC,GC,SNHS],NHC),
13      % Trigger rule if at least one change is effective
14      (NHs \== Hs ; NHS \== HS ; NHC \== HC) |
15      % NHc is the new min card for H
16      ord_int_length(NHs,SNHs),
17      NHc is max(Hc,SNHs),
18      set(H,support(NHs,NHS),card(NHc,NHC)).
```

This rule is triggered when it encounters: a set constraint between 3 sets ($H = F \cap G$ where H, F, and G are abstract names) in Line 1, the declarations of the sets F and G together with their domains (Lines 2 and 3), and the declaration of the set H (Line 4). The declaration of the set F is given by `set(F,support(Fs,FS),card(_Fc,FC))` where `F` is the name of the set, `support(Fs,FS)` gives the lower bound (`Fs`) and upper bound (`FS`) of F, and `card(Fc,FC)` gives its minimum cardinality (`Fc`) and maximum cardinality (`FC`). Lines 5 to 14 defines a guard (between `<=>` and `|`), *i.e.*, a condition that must be fulfilled to trigger the rule. Here, this guard specifies that either the lower bound, or the upper bound, or the maximum cardinality of H must effectively be modified to trigger the rule. We can see that the new upper and lower bounds and maximum cardinality of H are computed in the guard. In this rule, the constraint $H = F \cap G$, as well as the definitions of F and G cannot be altered. Only the declaration of the set variable H is effectively modified when the rule is triggered (the declaration of H in Line 4 appears after the backslash "\"). Thus, if the rule matches and is triggered, a new declaration of H (with at least its lower or upper bound or its maximum cardinality modified) is added to the store (line 18).

Some other rules can also be split. For example, Rules (15) and (16) can be specialized when the finite domain variable x is closed (*i.e.*, its domain is reduced to a singleton). In this case, some less costly tests (membership instead of inclusion) can be performed in the guard before applying the rules.

5.4. Dynamic reduction

When instances are enormous, our CHR implementation for constraint propagation becomes less and less efficient, and memory problems can appear (see, for example, large instances of the SGP problem in Section 8). We thus propose a way to bypass this problem: while we generate the CSP model with data, we also dynamically reduce it.

Each time we generate a new constraint of the CSP instance, the reduction is triggered. Hence, the reduced CSP instances that we obtain are the same, making propagation dynamically on the CSP model or applying it to the CSP instance. However, we have a gain of memory: by reducing sets and variables and removing useless constraints on the fly, we can reduce the instances sooner, and hence tackle larger instances.

The difference is even more evident with disjunctions by applying Rules (15) and (16) on the fly. Indeed, tautologies and false disjuncts are immediately removed dynamically. For example, consider the Social Golfer instance

Table 1: Impact of reduction on the Social Golfer instance 5-4-5.

	Non-reduced instances	Reduced instances
sets	26	26
variables	50	50
constraints	98	20
disjunctions of size 4	47 500	18 000
disjunctions of size 3	0	0
disjunctions of size 2	0	864

5-4-5 (see Section 7). Table 1 shows the difference between the reduced and the non-reduced instances in terms of the number of sets, variables, constraints, and disjunctions.

However, reduction discards the original CSP instances. This is also the case with *MiniZinc* (Nethercote et al., 2007; Stuckey et al., 2014), which generates *FlatZinc* instances that are already transformed by some reductions or modifications of variables. This may not be a problem, except if some more tools have to process the instances or if the user wants to trace the whole transformation chain.

6. Encoding rules

The main objective of encoding rules ($\Leftrightarrow_{enc}$) is to translate set constraints from the original CSP model into propositional satisfiability clauses. There must be a translation for each type of constraint; otherwise, a generated SAT instance would be a kind of relaxation of the original problem. It is worth noting that the rules presented in the following subsection can apply to reduced set constraints, but also non-reduced ones. We only present a selection of the complete set of encoding rules.

6.1. Finite domain variable

The finite domain variable encoding rule is designed to enforce the condition that every finite domain variable is associated with one and only one value from its corresponding domain:

$$v :: D_v$$

$$\Leftrightarrow_{enc}$$

$$\bigvee_{x \in D_v} (\wedge_{y \in D_v, x \neq y} (\neg y_v) \wedge x_v)$$

Thus, in CNF the rule becomes:

$$v :: D_v$$

$$\Leftrightarrow_{enc}$$

$$\bigwedge_{x,y \in D_v, x \neq y} (\neg x_v \vee \neg y_v) \wedge \bigvee_{x \in D_v} x_v \qquad \binom{|D_v|}{2} \text{ binary clauses and one } |D_v| - ary \text{ clause}$$

The number of generated Boolean variables is $|D_v|$.

6.2. Set variable

For encoding a set $F :: (\underset{.}{F}, \dot{F}, \lfloor F \rfloor, \lceil F \rceil)$, some Boolean variables have to be created, and all variables of the lower bound $\underset{.}{F}$ are set to true. For a given constant x from the universe, the creation of a variable expressing that it is part of a set F is denoted by x_F.

$$F :: (\underset{.}{F}, \dot{F}, \lfloor F \rfloor, \lceil F \rceil)$$

$$\Leftrightarrow_{enc}$$

$$\begin{cases} \forall x \in \dot{F}, a\ Boolean\ variable\ x_F\ is\ available & 0\ clauses \\ \bigwedge_{x \in \underset{.}{F}} x_F & |\underset{.}{F}|\ unit\ clauses \end{cases}$$

Thus, for each variable of the upper bound, we create a variable x_F. If it is also in the lower bound, we force it to be true by adding the unit clause x_F.

6.3. Set intersection constraint

To be extensive and clear, we must consider all the possible cases with regards to the sets $\dot{H}$, $\underset{.}{H}$, $\dot{F}$, $\underset{.}{F}$, $\dot{G}$, and $\underset{.}{G}$. We denote useless and impossible cases with the "–" symbol.

$$H = F \cap G$$

$$\Leftrightarrow_{enc}$$

$$\forall x \in \mathcal{U} \left\{ \begin{array}{ll}
x \in \underset{\cdot}{H} & \left\{ \begin{array}{ll}
x \in \underset{\cdot}{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & true & \\ x \in G^? & x_G & |\underset{\cdot}{H} \cap \underset{\cdot}{F} \cap G^?| \text{ unit clauses} \\ x \notin \dot{G} & false & \end{array} \right. \\
x \in F^? & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & x_F & |\underset{\cdot}{H} \cap F^? \cap \underset{\cdot}{G}| \text{ unit clauses} \\ x \in G^? & x_F \wedge x_G & |\underset{\cdot}{H} \cap (\dot{F} \setminus \underset{\cdot}{F}) \cap G^?| \text{ binary clauses} \\ x \notin \dot{G} & false & \end{array} \right. \\
x \notin \dot{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & false & \\ x \in G^? & false & \\ x \notin \dot{G} & false & \end{array} \right.
\end{array} \right. \\
x \in H^? & \left\{ \begin{array}{ll}
x \in \underset{\cdot}{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & x_H & |H^? \cap \underset{\cdot}{F} \cap \underset{\cdot}{G}| \text{ unit clauses} \\ x \in G^? & x_H \leftrightarrow x_G & |H^? \cap \underset{\cdot}{F} \cap G^?| \times 2 \text{ binary clauses} \\ x \notin \dot{G} & \neg x_H & |H^? \cap (\underset{\cdot}{F} \setminus \dot{G})| \text{ unit clauses} \end{array} \right. \\
x \in F^? & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & x_H \leftrightarrow x_G & |H^? \cap F^? \cap \underset{\cdot}{G}| \times 2 \text{ binary clauses} \\ x \in G^? & x_H \leftrightarrow x_F \wedge x_G & |H^? \cap F^? \cap G^?| \times 3 \text{ binary clauses} \\ x \notin \dot{G} & \neg x_H & |H^? \cap (F^? \setminus \dot{G})| \text{ unit clauses} \end{array} \right. \\
x \notin \dot{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & \neg x_H & |H^? \cap \underset{\cdot}{G} \setminus \dot{F}| \text{ unit clauses} \\ x \in G^? & \neg x_H & |H^? \cap G^? \setminus \dot{F}| \text{ unit clauses} \\ x \notin \dot{G} & \neg x_H & |H^? \setminus \dot{F} \setminus \dot{G}| \text{ unit clauses} \end{array} \right.
\end{array} \right. \\
x \notin \dot{H} & \left\{ \begin{array}{ll}
x \in \underset{\cdot}{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & false & \\ x \in G^? & \neg x_G & |\underset{\cdot}{F} \cap (G^? \setminus \dot{H})| \text{ unit clauses} \\ x \notin \dot{G} & - & \end{array} \right. \\
x \in F^? & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & \neg x_F & |F^? \cap \underset{\cdot}{G} \setminus \dot{H}| \text{ unit clauses} \\ x \in G^? & \neg x_F \vee \neg x_G & |F^? \cap G^? \setminus \dot{H}| \text{ binary clauses} \\ x \notin \dot{G} & - & \end{array} \right. \\
x \notin \dot{F} & \left\{ \begin{array}{lll} x \in \underset{\cdot}{G} & - & \\ x \in G^? & - & \\ x \notin \dot{G} & - & \end{array} \right.
\end{array} \right.
\end{array} \right.$$

6.4. Set cardinality constraint

The set cardinality constraint $x = |F|$ links the cardinal of the set F to the finite domain variable x. Some efficient encodings of this global constraint have already been proposed, see for example Abío et al. (2015, 2013) or Bailleux & Boufkhad (2003). The encoding presented in Bailleux & Boufkhad (2003) is based on the unary representation of integers[3] and the use of two essential elements.

The first one is a balanced binary tree called the *totalizer*. It associates an auxiliary output variable to each input variable involved in the cardinality constraint and permits to sort these new auxiliary variables by giving priority to those having a true value. A set of internal variables, called linking variables, is employed to associate the input and output variables. Each internal node N in the totalizer represents the union of its corresponding children, C^1 and C^2, which are sets of Boolean variables. The α-th variable of the set C^1 is denoted as C^1_α. Moreover, the input and output variables are in the leaves and the root of the binary tree, respectively. The following propositional satisfiability clauses are required to encode each node of the totalizer binary tree:

$$\bigwedge_{\substack{0\leq\alpha\leq|C^1|,\\ 0\leq\beta\leq|C^2|,\\ 0\leq\gamma\leq|N|,\\ \alpha+\beta=\gamma}} \left(\neg C^1_\alpha \vee \neg C^2_\beta \vee N_\gamma\right) \wedge \left(C^1_{\alpha+1} \vee C^2_{\beta+1} \vee \neg N_{\gamma+1}\right),$$

with $C^1_0 = C^2_0 = N_0 = 1$, and $C^1_{|C^1|+1} = C^2_{|C^2|+1} = N_{|N|+1} = 0$.

The second essential element of the encoding presented in Bailleux & Boufkhad (2003) is the *comparator*. It forces k to take a value equal to the cardinal of the set by fixing the value of the first k output variables of the totalizer binary tree, denoted as s_i. The following expression can be used to encode the comparator:

$$\bigwedge_{1\leq i\leq k} s_i \bigwedge_{k+1\leq j\leq n} \neg s_j$$

Suppose that $|\dot{F}| = n$, then the total number of clauses and variables generated by the constraint $|G| = k$ are computed as follows:

- $n + \sum_{i=1}^n 2u_i^n(\lfloor \frac{u_i^n}{2} \rfloor + 1)(\lceil \frac{u_i^n}{2} \rceil + 1) - (\frac{u_i^n}{2} + 1)$ clauses,
- $\sum_{i=1}^n u_i^n$ variables.

with $u_n^n = 1$, $u_1^n = n$ and $u_i^n = u_{2i-1}^n + 2u_{2i}^n + u_{2i+1}^n$.

6.5. Disjunction constraint

The disjunction constraint is a “meta-constraint” working with several constraints:

$$constraint_1 \vee \ldots \vee constraint_n$$

To generalize the encoding of the disjunction constraint into SAT, we reuse the classical transformations of the constraints into CNF. However, the raw encoding does not provide a formula in CNF. We have thus to apply the well-known De Morgan’s laws that increase the number of clauses exponentially. If each constraint $c \in C$ used in the disjunction is composed of n^c clauses, then $n = \prod_{c\in C} n^c$ clauses are generated by the encoding. However, this encoding generates the same number of variables as if the constraints were handled separately.

[3]An integer $k \in [0..n]$ is defined by a sequence of k ones followed by a sequence of $n - k$ zeros.

7. Models

In this section, we propose some models for two well-known problems: the Social Golfer Problem and the Sport Tournament Scheduling problem.

7.1. *Social golfer problem*

The Social Golfer Problem, listed as problem number 10 in the CSPLib (Harvey, 2019), is defined as follows. There exist q golfers that play every week during a period of w weeks. Golfers are split into g groups of p golfers, each one ($q = p \cdot g$). The objective of the SGP is to construct a schedule of play for the q golfers, such that no golfer plays in the same group as any other golfer more than once. It is a very attractive problem since SGP, and closely related problems arise in different research areas like encryption and covering problems (Hsiao et al., 1970; Stinson, 1994). Furthermore, there exist various SGP instances that are still open (Pegg, 2007).

A set constraint model for this problem can be easily obtained. The universe is the set of players, *i.e.*, the set containing the identification number of each golfer. To model the groups of golfers, we need $w \cdot g$ set variables.

- Universe (the golfers): $\mathcal{U} :: ([1, q], [1, q], q, q)$
- The set of golfers: $\mathcal{P} :: \mathcal{U}$
- g groups of p golfers for w weeks: $\forall i \in [1, w]$, $\forall j \in [1, g]$, $G_{i,j} :: (\emptyset,\ \mathcal{U},\ p,\ p)$

The constraints used to model the SGP are now detailed.

- There are exactly p golfers in each group on a given week:

$$\forall i \in [1, w],\ \forall j \in [1, g],\ |G_{i,j}| = p \tag{17}$$

  Note that this constraint is present in the initial definition of the SGP, but it is now redundant with the definition of $G_{i,j}$ variables.

- All the golfers play every week:

$$\forall i \in [1, w] \bigcup_{j \in [1,g]} G_{i,j} = \mathcal{P} \tag{18}$$

- During the same week, no one can play in two different groups:

$$\forall i \in [1, w] \bigcap_{j \in [1,g]} G_{i,j} = \emptyset \tag{19}$$

  Nevertheless, given that Constraint (19) is implied by Constraints (17) and (18), then it is not necessary.[4]

[4]Moreover, we did some experiments, and Constraint (19) does not allow more reduction and does not speed up propagation.

- A couple of golfers is not allowed to play together in two different weeks in the same group (socialization constraints):

$$\begin{aligned} &\forall w_1, w_2 \in [1, w], p_i, p_j \in \mathcal{P}, g_1, g_2 \in [1, g], \\ &\qquad w_1 > w_2 \;\wedge\; i > j \;\wedge \\ p_i \in G_{w_1,g_1} \;\wedge\; p_j \in G_{w_1,g_1} \;\wedge\; &p_i \in G_{w_2,g_2} \;\rightarrow\; p_j \notin G_{w_2,g_2} \end{aligned} \tag{20}$$

  Constraint (20) means: given a couple of golfers currently playing in a group g_1, if p_1 plays in another group g_2, then p_2 cannot play in this group g_2. We can alternatively define these constraints by using cardinality constraints in the following way:

$$\begin{aligned} &\forall w_1, w_2 \in [1, w], g_1, g_2 \in [1, g], w_1 > w_2 \;\wedge \\ &\qquad |G_{w_1,g_1} \cap G_{w_2,g_2}| \leq 1 \end{aligned} \tag{21}$$

- Symmetry breaking
  - The first week is fixed:

$$\forall i \in [1, p], p_i \in G_{1,((i-1)\ div\ q)+1} \tag{22}$$

  - The second symmetry breaking complements Constraint (22) by distributing the first group of p golfers (that have already played together during the first week) among distinct groups in the following weeks. Note that the inequality $g < p$ should always be satisfied; otherwise, there is no solution:

$$\forall i \in [2, w],\ \forall j \in [1, p], \quad p_j \in G_{i,j} \tag{23}$$

  - Ordering groups every week: groups can be ordered inside a week with respect to their first player. For the first p groups, Constraint (23) already does it.

$$\forall i \in [1, w], \forall j \in [1, g-1], \;\; min(G_{i,j}) < min(G_{i,j+1}) \tag{24}$$

  - Ordering weeks: weeks can be ordered with respect to the maximum element of each first group.

$$\forall i \in [1, w-1], \;\; max(i, 1) < max(i+1, 1) \tag{25}$$

7.2. Sports tournament scheduling problem

The Sports Tournament Scheduling problem, listed as problem number 26 in the CSPLib (Walsh, 2019), is defined as follows.

"The problem is to schedule a tournament of n teams over $n - 1$ weeks, with each week divided into $n/2$ periods, and each period divided into two slots. The first team in each slot plays at home, while the second plays the first team away. A tournament must satisfy the following three constraints: every team plays once a week; every team plays at most twice in the same period over the tournament; every team plays every other team."

From this definition, it is easy to observe that an instance of the STS problem is fully specified by the unique input parameter n (number of sportive teams to be scheduled). In the following, we introduce a set constraint model for a total of $w = n - 1$ weeks, each one of them divided into $p = n/2$ periods:

- Universe (the sportive teams): $\mathcal{U} :: ([1,n],[1,n],n,n)$
- The set of teams: $\mathcal{T} :: (\mathcal{U},\mathcal{U},n,n)$
- For every week of the tournament, and every single period, games are specified as sets of two teams: $\forall i \in [1,w],\ \forall j \in [1,p],\quad G_{i,j} :: (\emptyset,\mathcal{U},2,2)$

The constraints used to model the STS are now detailed.

- Each week of the tournament, each team must play:

$$\forall i \in [1,w],\ \mathcal{T} = \bigcup_{j\in[1,p]} G_{i,j} \tag{26}$$

- Each team must play at most twice during the same period of two different weeks:

$$\forall q \in [1,p],\ \forall i \in [1,w-2],\ \forall j \in [i+1,w-1],\ \forall k \in [j+1,w], G_{i,q} \cap G_{j,q} \cap G_{k,q} = \emptyset \tag{27}$$

- Each team must play every other team. As required above, in each of the $n - 1$ weeks of the tournament, each team must play a match; it is thus enough to constrain that two matches cannot be equal:

$$\forall i \in [1,w-1],\ \forall j \in [i+1,w],\ \forall p_1,p_2 \in [1,p],\ \ G_{i,p_1} \neq G_{j,p_2} \tag{28}$$

- Symmetry breaking
  - We can fill the first week as follows: Teams 1 and 2 play together during the first period; Teams 3 and 4 play during the second one, and so on:

$$\forall i \in [1,n],\ \ i \in G_{1,((i-1)\ div\ 2)+1} \tag{29}$$

  - Starting from the second week, we can place the first team in "diagonal" during p weeks:

$$\forall i \in [1,p],\ \ 1 \in G_{i+1,i} \tag{30}$$

8. Experimental results

In this section, we analyze the resolution of SAT instances corresponding to CSP instances of the SGP and STS problems translated by our $\Leftrightarrow_{enc}$ rules. These instances are named by the triple g_p_w for SGP and by the number of teams for STS. For SGP, we test and compare the two ways of modeling the socialization constraint of the problem: CARD and IMP models. The first one uses the cardinality Constraint (21) ("C") and the second one the implication Constraint (20) ("I").

For each instance of the studied problems (SGP and STS), the $\Rightarrow_{red}$ rules compute a reduced instance. Reduced instances for the STS problem are identified by adding the suffix "_R" to its corresponding name, while those of the SGP are labeled "I_R\D" (\D stands for treatment without the disjunction reduction rules proposed in Section 4.4). The use of dynamic reduction (see Section 5.4) corresponds to the line with "DynI_R".

The experiments presented in this work were run on a CPU Intel® Xeon® E5-2670 at 2.3 GHz, 16 GB of RAM with 64 bits Linux operating system (Ubuntu 18.04). For experiments with large instances of STS (Table 6), a huge RAM of 230 GB was available only to limit the tests by time. All times are CPU times in seconds. The $\Leftrightarrow_{enc}$ rules were coded in C++ and compiled with g++ (7.4.0) using the optimization flag -O3, while the $\Rightarrow_{red}$ rules were implemented as Constraint Handling Rules (CHR) (Frühwirth, 2009) in SWI-Prolog 7.6.2. The propositional satisfiability solver employed for all our experiments is *MiniSAT* (version 2.2).

8.1. Models

We have presented two different models for the SGP problem corresponding to two ways for treating the socialization constraints: one based on cardinalities of intersections of an exponential number of sets (the CARD model), and one based on implications, and thus disjunctions of constraints (the IMP model). Both models give the same solutions. We report two experiments to show the difference between the CARD and the IMP models. In Table 2, we illustrate our discussion by comparing two instances. The first column contains the name of the instance (groups_players-per-group_weeks). The second one provides the satisfiability of the instance (S for satisfiable, U for unsatisfiable). The third one explains the encoding method (C for cardinality, I for implication, C_R for cardinality with reductions, and I_R\D for cardinality with reductions without disjunction). The next column provides the reduction time (n/a when no reduction is performed). The last three groups of columns correspond to model characteristics for the SAT instance (number of clauses (#cl) and variables (#var)), the encoding time, and finally the solving time (just for the SAT instance and for the global process (reduction, encoding and solving)). Here only raw instances (noted as Unrefined) are observed to highlight the impact of model choice. Tests are realized with the 7_2_11 and 7_2_15 instances: the first one is satisfiable, and the second one is unsatisfiable (U). For the 7_2_11 instance, the difference between the raw instance solving times (C for the CARD model and I for the IMP model) and the reduced instances solving times (C_R and I_R\D[5]) are still affordable. For the 7_2_15 SGP, the difference is already significant (1083.63 seconds for

[5]As the CARD model did not use disjunction constraints, we only analyze the reduced instances without reduction rules corresponding to the disjunction constraints.

Table 2: CARD vs. IMP for 2 SGP instances: explosion of the number of variables for the CARD models

Inst.	SAT	Enc.	Reduction time	Model characteristics		Encoding time	Solving time	
			$\Rightarrow_{red}$	Unrefined		$\Leftrightarrow_{enc}$	Unrefined	
			sec.	#cl	#var	sec.	sec.	sum
7_2_11	S	C	n/a	307	43 008	1.03	0.10	1.13
		I	n/a	272 117	5 278	0.93	0,07	1.00
		C_R	0.92	238 676	30 426	0.82	0.38	1.82
		I_R\D	0.37	166 686	4 096	0.53	0,05	0.95
7_2_15	U	C	n/a	720 909	79 212	5.41	1 083.63	1 089.04
		I	n/a	504 819	7 182	1.64	249.82	251.46
		C_R	1.04	469 084	58 422	4.88	425.65	431.57
		I_R\D	0.66	324 562	5 712	1.08	54.70	56.44

the C model and 425.65 seconds for the C_R model). For larger instances, the CARD model cannot be solved (or even encoded) anymore.

Whereas the CARD model is competitive for small instances (both in terms of solving time and instance size), it quickly becomes intractable as instances grow. Indeed, the number of variables for encoding the cardinality constraints in SAT quickly explodes. We thus focus on the IMP model for the SGP problem in the following.

8.2. Efficiency of reduction rules and SAT preprocessing

A preprocessing can be used to decrease the size of CNF instances. Though those preprocessed instances are smaller than the original raw instances, they are not necessarily easier to solve. The preprocessing may remove easy to reach symmetrical solutions and keeps only farther solutions. It may also happen that the reduction changes the structure of the instance, and consequently changes the search space and position of solutions.

SatELite (Eén & Biere, 2005) is a CNF minimizer that we have used for SAT preprocessing. *SatELite* can be applied either as a simple initial unit propagation process (UP_{Sat}) or as a complete minimizer (CM_{Sat}) achieving subsumption, self-subsuming resolution, and elimination of variables using substitution. We now use the term "unit propagation" for *SatELite* applied as a simple unit propagation process, and "CNF minimizer" as its complete process. We compare these two different SAT reduction approaches over the two problems addressed in this paper (SGP and STS). Tables 3 and 4 summarize the results. The three first columns of Table 3 provide the instance name, the satisfiability of the problem (S for satisfiable, U for unsatisfiable, and ? when *MiniSAT* cannot provide an answer), and the encoding type. In Table 4, the first column provides the name. The next column in both tables is the reduction time (n/a when there is no reduction). The next three blocks of columns are split in three, corresponding to the different models (unrefined, preprocessed by unit propagation, and preprocessed by CNF minimizer). Each block corresponds respectively to the model characteristics (number of clauses and variables), the encoding time, and the solving time (with a sum column corresponding to the total running time of all the process). Bold values correspond to the best

Table 3: Results for SGP instances with the IMP model

				Model characteristics						Encoding or Preprocessing time			Solving time					
Inst.	SAT	Enc.	$\Rightarrow_{red}$	Unrefined		UP_{Sat}		CM_{Sat}		$\Leftrightarrow_{enc}$	UP_{Sat}	CM_{Sat}	Unrefined		UP_{Sat}		CM_{Sat}	
			sec.	#cl	#var	#cl	#var	#cl	#var	sec.	sec.	sec.	sec.	sum	sec.	sum	sec.	sum
6_2_12	U	I	n/a	175 894	4 068	96 250	1 672	83 060	1 122	0.57	1.81	3.19	2.61	3.18	7.67	10.05	1.43	5.19
		I_R\D	0.39	104 130	3 084	96 074	1 562	76 824	1 078	0.37	0.28	1.98	1.10	***1.86***	1.10	2.15	1.68	4.42
		DynI_R	2.31	104 130	3 084	96 074	1 562	82 958	1 078	0.40	0.23	1.86	54.02	56.73	2.53	5.47	42.05	46.62
6_2_13	U	I	n/a	205 992	4 404	114 840	1 824	99 100	1 224	0.49	2.27	5.15	7.82	8.31	3.54	6.30	2.72	8.36
		I_R\D	0.45	123 432	3 360	114 648	1 704	91 656	1 176	0.35	0.22	2.55	17.23	18.03	1.82	***2.83***	2.02	5.36
		DynI_R	2.73	123 432	3 360	114 648	1 704	98 854	1 176	0.38	0.28	1.98	1.93	5.04	17.71	21.10	82.94	88.02
6_5_6	U	I	n/a	280 981	6 498	123 500	2 905	123 500	2 905	0.36	5.10	5.56	-	-	-	-	-	-
		I_R\D	0.58	140 300	4 585	123 260	2 840	123 260	2 840	0.19	0.37	0.40	-	-	-	-	-	-
		DynI_R	3.94	140 300	4 585	123 260	2 840	123 260	2 840	0.16	0.31	0.31	-	-	2 819.91	2 824.32	2 467.53	***2 471.94***
7_2_13	S	I	n/a	379 550	6 230	225 144	2 616	200 771	1 824	1.30	5.39	8.32	0.10	1.40	0.04	6.73	0.07	9.69
		I_R\D	0.46	239 108	4 904	224 796	2 472	184 464	1 680	0.55	0.62	3.71	0.08	***1.10***	0.04	1.68	0.06	4.79
		DynI_R	5.82	239 108	4 904	224 796	2 472	194 152	1 680	1.20	0.56	5.85	0.08	7.11	0.07	7.66	0.06	12.93
7_2_15	U	I	n/a	504 819	7 182	308 280	3 052	274 705	2 128	1.64	13.72	20.58	249.82	251.46	1 205.26	1 220.62	436.83	459.05
		I_R\D	0.66	324 562	5 712	307 874	2 884	252 560	1 960	1.08	0.92	9.02	54.70	56.44	234.23	236.89	411.10	421.86
		DynI_R	8.81	324 562	5 712	307 874	2 884	265 659	1 960	0.74	0.78	8.35	675.06	684.60	39.79	***50.11***	509.21	527.10
7_4_7	U	I	n/a	445 154	8 120	221 508	3 672	220 525	3 258	2.17	12.61	11.37	-	-	-	-	-	-
		I_R\D	0.57	244 180	5 992	221 172	3 576	179 203	2 808	2.25	0.73	4.60	-	-	-	-	-	-
		DynI_R	5.19	244 180	5 992	221 172	3 576	217 424	2 904	2.46	0.58	1.46	-	-	-	-	1 671.73	***1 680.84***
7_7_7	?	I	n/a	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
		I_R\D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
		DynI_R	23.76	745 885	11 956	682 521	7 728	682 521	7 728	0.99	1.72	1.68	-	-	-	-	-	-
8_4_4	S	I	n/a	237 324	6 240	84 900	2 436	84 876	2 400	3.30	3.13	3.04	0.03	3.33	0.02	6.45	0.02	6.36
		I_R\D	0.32	102 668	4 148	84 648	2 388	72 035	2 088	1.78	0.21	0.84	0.02	2.12	0.03	2.33	0.02	2.95
		DynI_R	1.35	102 668	4 148	84 648	2 388	84 112	2 160	0.40	0.24	0.45	0.04	***1.79***	0.03	2.03	0.03	2.23
8_4_5	S	I	n/a	375 984	7 776	161 808	3 248	161 776	3 200	1.01	7.89	7.22	0.05	1.06	0.06	8.96	0.03	8.27
		I_R\D	0.46	185 456	5 488	161 472	3 184	136 591	2 784	0.45	0.32	1.54	0.03	***0.94***	0.03	1.26	0.05	2.50
		DynI_R	3.1	185 456	5 488	161 472	3 184	160 475	2 880	0.72	0.43	0.70	0.06	3.89	0.06	4.31	0.06	4.58
8_4_6	S	I	n/a	546 388	9 312	263 020	4 060	262 980	4 000	1.50	9.45	9.46	0.09	***1.59***	0.06	11.01	0.06	11.02
		I_R\D	0.61	292 548	6 828	262 600	3 980	221 420	3 480	0.93	0.85	5.11	0.11	1.65	0.09	2.48	0.08	6.73
		DynI_R	5.68	292 548	6 828	262 600	3 980	261 227	3 600	0.78	0.71	0.74	0.11	6.57	0.09	7.27	0.07	7.27
8_4_7	S	I	n/a	748 536	10 848	388 536	4 872	388 488	4 800	1.99	16.12	14.67	0.68	2.67	0.22	18.33	0.77	17.43
		I_R\D	0.75	423 944	8 168	388 032	4 776	326 519	4 176	1.05	1.12	8.04	0.49	***2.28***	0.17	3.09	0.53	10.37
		DynI_R	8.43	423 944	8 168	388 032	4 776	385 980	4 320	2.08	0.91	1.73	0.29	10.80	0.40	11.82	0.14	12.39

Table 4: Efficiency of SAT preprocessing - STS

		Model characteristics						Encoding or Preprocessing time			Solving time					
Inst.	$\Rightarrow_{red}$	Unrefined		UP_{Sat}		CM_{Sat}		$\Leftrightarrow_{enc}$	UP_{Sat}	CM_{Sat}	Unrefined		UP_{Sat}		CM_{Sat}	
	sec.	#cl	#var	#cl	#var	#cl	#var	sec.	sec.	sec.	sec.	sum	sec.	sum	sec.	sum
8	n/a	23 812	5 528	13 794	3 867	6 422	1 040	0.00	0.04	0.28	0.01	**0.01**	0.01	0.05	0.01	0.29
8_R	0.08	16 606	4 549	13 345	3 902	2 649	426	0.00	0.02	0.26	0.00	**0.08**	0.00	0.10	0.00	0.34
10	n/a	77 135	17 170	46 917	12 530	29 233	5 194	0.00	0.12	0.91	3.37	3.37	1.18	**1.30**	5.32	6.23
10_R	0.11	58 235	14 788	45 918	12 650	13 729	2 352	0.00	0.07	0.74	0.01	**0.12**	0.01	0.19	-	-
12	n/a	198 198	42 612	124 980	32 297	85 581	15 576	0.02	0.40	3.08	143.96	**143.98**	509.24	509.64	266.26	269.34
12_R	0.22	148 364	35 188	112 467	29 541	40 188	6 624	0.02	0.22	2.22	0.04	**0.28**	0.03	0.49	0.01	2.47
14	n/a	436 541	91 154	280 964	70 707	205 706	38 492	0.04	0.76	8.05	-	-	-	-	-	-
14_R	0.41	342 805	78 104	257 290	65 772	93 320	15 339	0.03	0.42	5.42	0.22	**0.66**	0.06	0.92	0.04	5.90

values. Numerical values are replaced by the symbol "-" when an instance is not solved (because of time-out). Note that we refer to the model produced by the $\Leftrightarrow_{enc}$ encoding procedure as the unrefined one, and the total running time is limited to 3600 seconds.

8.2.1. Reduction rules

Tables 3 and 4 show that reduction rules decrease the final size of the encoded instances. However, these reduction rules may provide an extra cost in terms of computational solving time. In the case of certain SGP instances (mainly those encoded by DynI_R) and all small STS instances, the application of the reduction rules seems to consume most of the total running time needed for solving an instance. Furthermore, shorter solving times were also observed over these reduced instances (most of instances reduced by I_R\D and DynI_R for SGP, and all instances suffixed by _R for STS). Recall that SGP models use disjunction constraints which do not appear in STS models. For large SGP instances that are reduced by all the reduction rules except those for disjunction (I_R\D), the reduction process is even more interesting in terms of total solving times.

Reduction rules for disjunction constraints are time-consuming and do not help to provide faster results. For large instances, disabling the specific reduction rules for disjunction permits us to reach a good trade-off between the final size of the preprocessed SAT instance and the total solving time. Hence, only rules for reducing variable domains are triggered, and the SAT solver solves the disjunctions remaining in the encoded instances.

8.2.2. SAT preprocessing

Based on the results presented in Tables 3 and 4, we now analyze the effect of SAT preprocessing on the resolution chain. In Table 5, we also summarize the comparisons of the non-reduced and unrefined instances (I - Unrefined) with all the other generated instances (I - UP_{Sat}, I - CM_{Sat}, I_R\D - Unrefined, I_R\D - UP_{Sat}, I_R\D - CM_{Sat}, DynI_R -

Table 5: Percentage of reduction in terms of clauses, variables, and solving time w.r.t. the unrefined instances without reduction (grey cells).

		Unrefined			UP_{Sat}			CM_{Sat}		
		cl	var	time	cl	var	time	cl	var	time
		avg. % (s.d.)	avg. % (s.d.)	+;=;- + %;= %;- %	avg. % (s.d.)	avg. % (s.d.)	+;=;- +%;=%;-%	avg. % (s.d.)	avg. % (s.d.)	+;=;- + %;= %;- %
SGP	I	▨	▨	▨	49.66 (7.87)	57.37 (1.97)	1;3;7 9;27;64	52.50 (5.49)	63.41 (7.17)	0;3;8 0;27;73
	I_R\D	44.60 (6.59)	25.96 (4.00)	6;3;2 55;27;18	49.74 (7.88)	58.93 (2.37)	4;3;4 36;28;36	57.82 (5.73)	66.91 (6.01)	2;3;6 18;27;55
	DynI_R			2;3;6 18;27;55			3;2;6 27;18;55	53.09 (5.13)	66.29 (6.37)	3;1;7 27;9;64
STS	I	▨	▨	▨	38.46 (2.82)	25.93 (3.33)	1;1;2 25;25;50	61.21 (8.74)	68.04 (10.04)	0;1;3 0;25;75
	I_R	25.92 (4.71)	15.83 (2.02)	3;0;1 75;0;25	42.19 (1.68)	28.56 (1.89)	3;0;1 75;0;25	82.36 (4.60)	86.56 (4.04)	2;0;2 50;0;50

Unrefined, DynI_R - UP_{Sat}, and DynI_R - CM_{Sat}).

Analysis based on Tables 3 and 4.

In Tables 3 and 4, it can be observed that unit propagation, as well as the CNF minimizer, are able to significantly decrease the size (in terms of the number of variables and clauses) of the analyzed unrefined instances. For both unrefined and reduced instances (*i.e.*, the _R instances), the application of unit propagation seems to provide instances of similar size.

As can be seen, the CNF minimizer always returns smaller or equal instances than unrefined or UP_{Sat} instances. Therefore, we could deduce that CSP reduction rules do not avoid encoding numerous redundant or subsumed clauses.

Although the CNF minimizer and unit propagation reduce the size of instances, we mainly obtain the best resolution times without SAT preprocessing (either UP_{Sat} or CM_{Sat}).

Analysis based on Table 5.

Table 5 proposes two types of analyses:

- Model characteristics: the average percentages of reduction (and the standard deviation into brackets) in terms of clauses and variables.
- Solving time: concerning the unrefined non-reduced instances (grey cells), the triplets "+;=;-" represent the number of instances respectively improving, equalizing, and deteriorating the solving time of the different solving chains. The cells "+%;=%;-%" are similar but in terms of percentages of instances.

Note that values presented in Table 5 include, in particular, the results of Tables 3 and 4. They also include the broad set of experiments we realized on the SGP and STS problems but that we do not present in this paper.

Table 5 shows that reduction associated with SAT preprocessing reduces a lot the number of clauses and variables. Instances I_R\D - CM_{Sat} for SGP and instances I_R - CM_{Sat} for STS produce the best reduction. However, the solving times of these instances are longer than for the unrefined instances without reduction (I - Unrefined). For the SGP instances I_R\D - CM_{Sat}, only 2 improve, 3 equalize, and 6 deteriorate. For the I_R - CM_{Sat} STS instances, only 2 improve the solving time, and the others deteriorate it.

The best results are obtained by instances I_R\D - Unrefined for SGP, where 6 over the 11 instances improve the solving times. For the STS problem, instances I_R - Unrefined and I_R - UP_{Sat} improve 3 times and deteriorate only once. Unit propagation is very cheap to apply. However, it is a standard process of classic SAT solvers. Thus, it is not useful to apply it twice (*i.e.*, as a preprocessing and in the SAT solver).

For efficiency reasons, it seems better only to apply our CSP reduction, since the SAT preprocessing seems to be too expensive compared to the gain of solving time.

8.3. Large instances

Table 3 shows that some large instances (in terms of groups, players, and weeks) of the SGP could not be solved. In this section, the analysis focuses on solving large instances of the STS problem by applying our reduction rules.

Table 6 discloses the significance of our CSP reduction rules for propositional satisfiability encoding. The first column provides the instance name. The next two blocks correspond to non-reduced instances, followed by reduced instances. For the first block, the columns correspond to the number of clauses, the number of variables, the encoding time, the *MiniSAT* solving time (limited to 7200 seconds), and the total solving time. For the reduced instance block, we have the same columns preceded by the reduction time (also limited to 7200 seconds). Instances up to a size of 66 teams can be solved after applying our reduction rules, while the larger initial instance for which a solution can be computed has only 12 sportive teams (c.f. Table 4). In our experiments, some larger instances (68 to 72) could be encoded into SAT, but *MiniSAT* was unable to solve them.

As said in Section 8.2, it may be more complicated to solve smaller reduced instances than larger non-reduced ones. However, the CSP reduction rules proposed in this paper reduce the size of the instances, but they also seem to simplify the problem structure. For example, the non-reduced instance 16 of the STS problem is smaller than the reduced instance 18, but it is not solved, whereas instance 18 is solved.

MiniSAT immediately quits the solving process for instances from 68 to 72: this is due to the excessive total number of clauses and variables. Some *ad hoc* solvers were specially designed for this problem. For example, in (Hamiez & Hao, 2014), the authors over-constrained the problem by adding extra (eventually non-redundant) constraints. They removed some symmetrical, but also some non-symmetrical solutions which help them to solve instances up to a size of 70 teams. Nevertheless, we have observed that half of the instances became unsatisfiable because of the additional constraints used to remove non-symmetrical solutions.

Table 6: Results produced by applying CSP reduction rules to a set of large instances of the STS problem.

Inst.	Non-reduced					Reduced					
	Unrefined		$\Leftrightarrow_{enc}$	Solving		$\Rightarrow_{red}$	Unrefined		$\Leftrightarrow_{enc}$	Solving	
	#cl $\times 10^3$	#var$\times 10^3$	sec.	sec.	sum	sec.	#cl$\times 10^3$	#var$\times 10^3$	sec.	sec.	sum
14	437	91	0.04	-	-	0.41	343	78	0.03	0.22	**0.66**
16	861	175	0.09	-	-	0.66	700	154	0.05	0.45	**1.16**
18	1 569	314	0.17	-	-	0.98	1 307	281	0.14	0.89	**2.01**
20	2 676	528	0.32	-	-	1.34	2 275	479	0.24	1.69	**3.28**
22	4 331	842	0.49	-	-	1.92	3 742	773	0.41	3.43	**5.76**
24	6 713	1 289	0.75	-	-	2.65	5 879	1 194	0.66	4.82	**8.13**
26	10 041	1 905	1.15	-	-	3.71	8 890	1 779	0.98	9.24	**13.93**
28	14 567	2 735	1.62	-	-	5.34	13 020	2 569	1.37	12.37	**19.08**
30	20 591	3 827	2.26	-	-	6.13	18 551	3 616	2.03	20.20	**28.36**
32	28 453	5 241	3.18	-	-	8.11	25 813	4 974	2.92	31.93	**42.96**
34	38 581	7 059	4.00	-	-	10.62	35 203	6 719	3.89	36.17	**50.68**
36	51 396	9 343	5.54	-	-	12.67	47 151	8 925	5.04	83.38	**101.09**
38	67 397	12 178	7.26	-	-	14.96	62 129	11 668	6.64	124.25	**145.85**
40	87 144	15 655	9.29	-	-	12.78	80 678	15 041	8.44	114.74	**135.96**
50	266 070	46 637	27.93	-	-	36.93	250 325	45 254	27.38	880.50	**944.81**
52	323 676	56 497	33.62	-	-	46.99	305 266	54 901	31.79	1 240.79	**1 319.57**
54	390 835	67 948	40.61	-	-	69.64	369 437	66 116	39.13	1 492.06	**1 600.83**
56	468 687	81 175	48.36	-	-	60.58	443 952	79 083	46.24	1 803.75	**1 910.57**
58	558 459	96 375	67.49	-	-	60.19	530 012	93 996	55.61	2 362.21	**2 478.01**
60	661 467	113 760	70.66	-	-	78.28	628 906	111 067	64.76	2 952.74	**3 095.78**
62	779 123	133 556	79.58	-	-	87.63	742 017	130 519	77.08	3 367.36	**3 532.07**
64	912 933	156 005	95.06	-	-	100.48	870 823	152 592	92.14	4 903.03	**5 095.65**
66	1 064 779	181 500	109.51	-	-	115.36	1 017 067	177 625	105.45	5 247.73	**5 468.54**
68	1 236 149	210 203	126.70	-	-	135.61	1 182 405	205 875	122.12	-	-
70	1 428 864	242 406	145.71	-	-	154.44	1 368 535	237 589	139.81	-	-
72	1 644 854	278 418	172.92	-	-	182.35	1 577 355	273 071	167.62	-	-

8.4. *Treating disjunctions: I_R vs. DynI_R and CSP vs. encoding*

In Table 7, we show some comparative executions of two ways of treating disjunctions in the CSP instances: I_R (instantiation, then reduction) and DynI_R (instantiation with dynamic reduction), as described in Section 5.4. They lead to the same CSP_R instances (see Figure 1). Based on Table 7, we can draw the following conclusions:

- The reduction time of DynI_R is much shorter than the one of I_R. When instances are larger (increasing either the number of groups, weeks, or players), the difference can be of several orders of magnitude.
- I_R cannot be used for large instances.

- The only drawback of DynI_R is that the raw instance is never generated; however, the raw instance could easily be generated using the modeler, and this is a very cheap task.

To summarize, the DynI_R approach is much more efficient and faster than I_R.

Table 7: SGP: I_R vs. DynI_R

	Reduction $\Rightarrow_{red}$		Model characteristics					
Inst	I_R	DynI_R	Unrefined		UP_{Sat}		CM_{Sat}	
	sec.	sec.	#cl	#var	#cl	#var	#cl	#var
4_3_2	0.10	0.00	793	218	362	98	287	65
4_3_3	3.30	0.00	2 141	388	1 327	196	1 100	130
4_3_4	22.38	0.00	4 092	558	2 895	294	2 408	195
4_3_5	86.59	0.06	6 646	728	5 066	392	4 256	260
4_3_6	151.93	0.11	9 803	898	7 840	490	6 613	325
4_4_3	17.47	0.01	3 624	560	2 384	312	2 334	256
4_4_4	92.04	0.04	7 060	808	5 232	468	5 138	384
4_4_5	343.23	0.13	11 600	1 056	9 184	624	9 054	512
4_4_6	501.90	0.58	17 244	1 304	14 240	780	14 059	640
4_4_7	997.76	0.36	23 992	1 552	20 400	936	20 160	768
5_2_7	333.58	0.20	15 124	1 072	12 660	540	9 463	318
5_2_8	574.15	0.31	20 144	1 244	17 276	630	12 917	371
5_2_9	869.15	0.26	25 880	1 416	22 608	720	16 954	424
5_2_10	1 197.05	0.37	32 332	1 588	28 656	810	21 499	477
5_2_11	1 862.15	0.61	39 500	1 760	35 420	900	26 634	530
5_3_5	472.36	0.19	16 016	1 252	12 920	712	11 298	504
5_3_6	1 201.21	0.36	24 220	1 550	20 365	890	18 004	630
5_3_7	1 685.81	0.54	34 110	1 848	29 496	1 068	26 283	756

Now, let us compare handling disjunctions at the level of CSP instances (see Section 5.4) or during the encoding (see Section 6.5). In Table 3, we can see that disjunction constraint reduction rules (both I_R and DynI_R) do not reduce the number of clauses. However, these rules reduce the size of some clauses as we show in this section. In-depth analyses of SGP instances were done, and all provided similar results. In Table 8, we illustrate these results with SGP instance 5-4-5. Each column corresponds to a different processing before encoding. There is a small difference between instances when treating disjunctions at the CSP level (I_R and DynI_R) or during the encoding (I_R\D). The

Table 8: Complete analysis of SAT encoding for the SGP instance 5-4-5.

	I	I_R	I_R\D	DynI_R
Variable Count	2 760	1 792	1 792	1 792
Literals	230 396	97 200	98 928	97 200
Positive Literals	19 346	11 216	11 216	11 216
Negative Literals	211 050	85 984	87 712	85 984
Clause Count	63 096	28 064	28 064	28 064
1-ary clauses	596	432	432	432
2-ary clauses	5 400	4 256	3 392	4 256
3-ary clauses	9 500	5 312	5 312	5 312
4-ary clauses	47 500	18 000	18 864	18 000
6-ary clauses	100	64	64	64

encoding can benefit from the CSP domain reduction to detect tautologies and contradictions as well, except some few cases that would require more work to be detected. For example, in Table 8, the encoding generates 864 clauses that are 4-ary with I_R\D and only 2-ary with I_R or DynI_R. We can see this difference in the number of literals (here only negative literals), which differs by 1728 (2×864) between I_R (or DynI_R) and I_R\D.

In terms of CPU time, as can be seen in Table 7, dealing with disjunction during encoding is much faster. However, it can be interesting to manage the disjunction during the CSP reduction: this is the case when SAT encoded instances become too large to be read by the SAT solver, see for example the 7_4_7 SGP instance in Table 3: only the CSP treatment of disjunctions enable to generate a SAT instance that can be read and solved using the *SatELite* preprocessing and *MiniSAT*.

8.5. Which is the "best" solving combination?

To summarize, for both the STS and SGP problems, the "best" solving combination consists in reducing variable domains at the CSP level, in handling disjunctions during the encoding, and to avoid the use of SAT preprocessing. Figure 3 shows this solving chain (it is one path of Figure 1). Remember that the different processes do not modify the solution space of the problem. However, since we keep only the first solution returned by the solving chain, each branch can lead to a different solution in Figure 1.

9. Comparisons with previous works

In comparison with our previous published work (Lardeux et al., 2015), the advantages of the approach presented in this paper are:

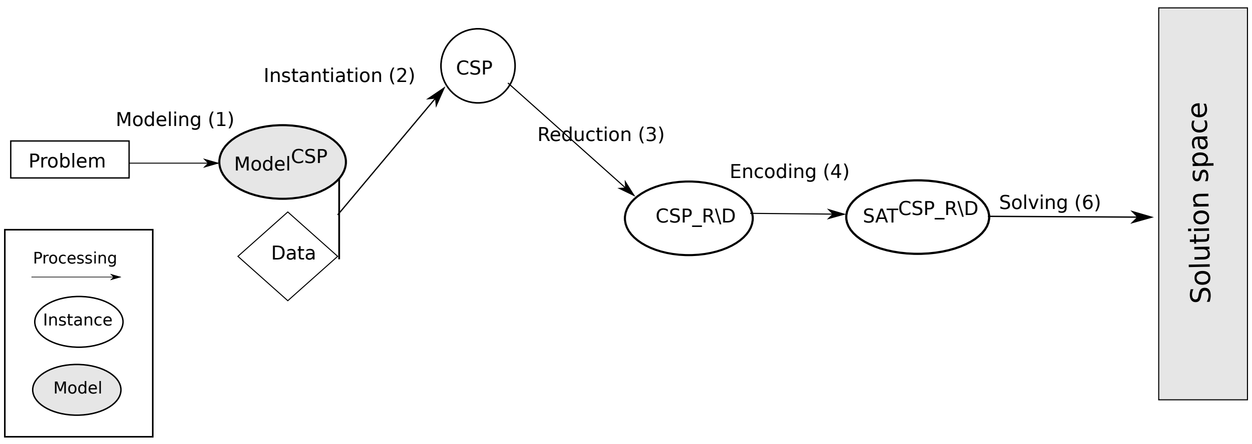

Figure 3: Best solving chain.

- The modeling language has been enhanced and enables the use of finite domain variables for modeling problems. Finite domain variables can now be associated with set variables through the use of the minimum, maximum, and cardinality constraints. It is an improvement of the language both for expressiveness, intuitiveness, and facility of use.

- The constraint propagation process became stronger than before. It produced smaller search spaces thanks to two new additions in the reduction rules: a) the use of upper and lower bounds of sets, and b) the implementation of minimum and maximum cardinalities of sets.

- Handling disjunctions now enables us to remove some complete tautologies and certain contradictions inside disjunctions. Consequently, the generated SAT instances contain a smaller number of clauses.

- The new $\Leftrightarrow_{enc}$ encoding rules became more general than the former ones. Moreover, they may apply to reduced as well as original raw CSP set constraints. Finally, they can produce SAT instances of a much smaller size.

To sum up, more problems can now be modeled, they can be solved more efficiently, and finally, larger instances can be tackled.

Let us now make some comparisons concerning other existing techniques for SAT encoding from the literature reported in Bacchus (2007) and Bessière et al. (2004). These works link the solving process of both CSP and SAT for some properties (*e.g.*, local consistencies for finite domain CSP). In contrast, the present work considers a distinct class of constraints (set constraints). Some solving chains are designed to produce SAT instances that are as small as possible, and as suitable as possible to be efficiently solved by classic SAT solvers. Moreover, the works reported in Bacchus (2007) and Bessière et al. (2004) lack of reduction mechanisms similar to our $\Rightarrow_{red}$ rules presented above.

The approach presented in Lardeux et al. (2009) is analogous to the methodology that we propose in this paper. Lardeux et al. (2009) propose to expressively handle the *alldifferent* global constraints, as well as the *overlapping* *alldifferent* constraints before using rewrite rules to translate them into SAT instances. However, in Lardeux et al. (2009), only those constraints related to the *alldifferent* constraint are treated.

To achieve the encoding of the *cardinality* global constraint into SAT, we used the results reported by Bailleux & Boufkhad (2003).

By using cardinality as proposed by Azevedo (2002), our constraint propagation process is stronger than the one of *Conjunto* (Gervet, 1994). In Correas et al. (2018), the authors introduced some new types of constraints linking finite domain variables and set variables. The interest of these constraints is double: they allow more expressiveness as shown for scheduling problems, and they make finite domains and set domains cooperate by reducing the search space and improving the efficiency of the existing set solvers for some specific cases. We thus also followed the work of Correas et al. (2018) by introducing constraints for connecting the minimum and maximum elements of a set with the set itself. The result is a more expressive language and some more powerful reductions. They are handy for breaking symmetries and for obtaining some more powerful reduction rules, both for sets and finite domains. In Correas et al. (2018), the authors show how to model some scheduling problems elegantly and how efficiently they are solved much faster than without these domain connecting constraints. Correas et al. (2018) also introduce a constraint enforcing all the elements of a set to be smaller than all the elements of a second set. This constraint is comparable to a global constraint (thus, with more reduction capacities) linking all the elements of the two sets. This constraint is also very powerful, and we plan to integrate it into our language in the future.

It is possible to obtain stronger reduction mechanisms (*e.g.*, Yip & Van Hentenryck, 2011) by using the *length-lex order* together with enhanced set representations. Even though the worst-case complexity of this class of constraint propagation methods is worse than for bound consistency, the *length-lex order* shows better performance than bound consistency over specific benchmarks (Yip & Van Hentenryck, 2011). Nevertheless, in our work, we aim at obtaining a good reduction-genericity trade-off for our technique. Moreover, the *length-lex order* is stronger for a complete solver interleaving reduction and enumeration, but in our work, we only need reduction.

In Hawkins et al. (2005), the authors employ Reduced Ordered Binary Decision Diagrams to compile set constraints. This structure is then exploited in the corresponding order to solve the problem at hand. It seems to be an efficient method, and the authors claimed that it could be adapted for working with multi-sets and integer numbers. In our case, preserving the constraint structures is of the utmost importance to guarantee that these constraints can be handled with other existing tools different from the ones presented in this paper. Thus, our further work also consists of adding additional global constraints that do not require any specific internal structure. Finally, as already said, we are not interested in implementing a solver, but at transforming and making suitable models and instances to generate better SAT instances to be solved by a classic SAT solver.

Concerning efficiency, solving the STS problem with our method and tools is a competitive alternative to the best (as we know) *ad hoc* solver (Hamiez & Hao, 2014). The largest instance size solved by this *ad hoc* STS solver is 70, whereas our tools only reached one of size 66. In terms of solving time, our method is faster. However, experimental conditions are different (CPU Intel® Xeon® E5-2670 at 2.3 GHz, 16 GB of RAM with 64 bits Linux operating system vs. CPU Intel® PIV processor at 2 GHz, 2 GB of RAM with 64 bits Linux operating system). For instance, the STS problem of size 66 is solved in 7500 seconds by the *ad hoc* solver and 5500 seconds by our tools. Note that this *ad hoc* solver was specifically and exclusively realized for the STS problem. Additionally, it consists of over-constraining the problem, and thus, some instances may become unsatisfiable.

We have shown in Lardeux et al. (2015) that our technique is competitive with hand-written SAT instances (Triska & Musliu, 2012) for the SGP problem. With our new method and tools, our generated SAT instances are even smaller and solved more quickly with *MiniSAT* than in Lardeux et al. (2015).

Although our goal was not to compete with solvers off-the-shelf, we ran different classic constraint solvers for the SGP problem. We gave our CSP set models to the *Conjunto* solver (Gervet, 1994). Only straightforward instances could be solved; for larger instances, it seems that the enumeration phase is a problem (with our system, we do not have this problem in the constraint programming part since the SAT solver is in charge of it). We also gave the same CSP set model to the well-known *MiniZinc* solver with its standard solver. It also got stuck quite quickly when instance size grows. We have tried other models with *MiniZinc*, *i.e.*, models consisting of finite integer domains issued from the *MiniZinc* GitHub.[6] Once again, the results were not better than the ones we obtained and reported in this paper. Thus, our approach can be considered competitive with classic CSP solvers applied to the SGP.

10. Summary of the method and conclusions

In this paper, we presented a method and some tools to reduce and encode set constraints into SAT instances. We can sketch the process as follows:

1. The modeling is achieved using set constraints with a very high expressiveness and declarativity.
2. By adding data to the model, we obtain CSP instances.
3. The application of our $\Rightarrow_{red}$ rules reduces CSP instances.
4. Reduced CSP instances are automatically encoded ($\Leftrightarrow_{enc}$) into SAT instances.
5. SAT instances may eventually be preprocessed.
6. Finally, a classic SAT solver solves the (preprocessed) SAT instances.

The proposed solution approach was illustrated in detail with two different hard problems: the SGP and the STS. We obtained some good results by applying reduction and encoding rules. For the STS problem, the results of our solution approach almost match those of the best-known solver designed explicitly for this problem (Hamiez

[6]https://github.com/MiniZinc/minizinc-benchmarks

& Hao, 2014). Furthermore, unlike this existing *ad hoc* solver, our solution approach prevents losing any solutions (unsatisfiable instances) by avoiding the production of over-constrained models.

We can observe several advantages when using our method and tools:

- Compared with direct SAT modeling, the proposed approach offers higher expressiveness, is relatively simpler to apply, and is independent from the selected solver (considering both CSP and SAT solvers).

- Our approach is less susceptible to include errors than direct hand-written propositional satisfiability encodings.

- CSP instances are made smaller (reduction of the search space, and withdrawal of useless constraints) with a propagation process applying reduction both on set and finite domain variables. We also treat disjunctions by removing some tautologies (useless constraints and tautology disjunctions), and contradictions inside disjunctions.

- The automatically generated SAT instances contain a smaller number of clauses and variables. They can undergo a preprocessing (*e.g.*, with *SatELite*) before resolution in a standard SAT solver (*MiniSAT* in our case). From our experiments, it appeared that these automatically generated SAT instances are suitable for being solved by *MiniSAT*.

- Adding our reduction process to the generation in most cases produces a total running time (which includes the reduction time, the encoding time, as well as the resolution time) smaller than that for non-reduced instances.

Although we have already pushed back the limits of our system in this paper, some limitations still exist:

- We have introduced some new constraints for finite domain variables, but some arithmetic constraints are still missing. In the future, we will focus on complementing the language with finite domain arithmetic constraints. To achieve it, a series of new reduction rules and encoding rules are necessary.

- Although powerful, our reduction process may sometimes not be worth it with regards to the total solving time. We thus plan to speed it up. To this end, we have already started some work about more powerful reductions for constraints such as the partitioning constraint. We are also considering special reductions for some patterns of conjunctions of set constraints. Last but not least, a new implementation in a language faster than CHR could also be considered.

- Encoding can still lead to too huge SAT instances, and thus it may cause memory problems. We think of several cardinality constraint encodings, each one with some specific advantages.

- The last point concerns the syntax of our models. Our XML-like syntax is convenient because we can enlarge it each time we want to try a new constraint. However, it would be interesting to use a standard format (such as XCSP3 (Boussemart et al., 2017)) to re-use already designed models.

Acknowledgments

The Mexican Secretariat of Public Education has partially supported the research of the third author under Grant SEP-CINVESTAV (2019-2020) No. 00114.

References

Abío, I., Mayer-Eichberger, V., & Stuckey, P. J. (2015). Encoding Linear Constraints with Implication Chains to CNF. In G. Pesant (Ed.), *Principles and Practice of Constraint Programming* (pp. 3–11). Springer International Publishing. doi:`10.1007/978-3-319-23219-5_1`.

Abío, I., Nieuwenhuis, R., Oliveras, A., & Rodríguez-Carbonell, E. (2013). A parametric approach for smaller and better encodings of cardinality constraints. In C. Schulte (Ed.), *Principles and Practice of Constraint Programming* (pp. 80–96). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/978-3-642-40627-0_9`.

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press. doi:`10.1017/CBO9780511615320`.

Azevedo, F. (2002). *Constraint Solving over Multi-valued Logics - Application to Digital Circuits*. Ph.D. thesis Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

Azevedo, F. (2007). Cardinal: A finite sets constraint solver. *Constraints*, *12*, 93–129. doi:`10.1007/s10601-006-9012-6`.

Bacchus, F. (2007). GAC Via Unit Propagation. In C. Bessière (Ed.), *Principles and Practice of Constraint Programming – CP 2007* (pp. 133–147). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/978-3-540-74970-7_12`.

Bailleux, O., & Boufkhad, Y. (2003). Efficient CNF Encoding of Boolean Cardinality Constraints. In F. Rossi (Ed.), *Principles and Practice of Constraint Programming – CP 2003* (pp. 108–122). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/978-3-540-45193-8_8`.

Bessière, C., Hebrard, E., & Walsh, T. (2004). Local Consistencies in SAT. In E. Giunchiglia, & A. Tacchella (Eds.), *Theory and Applications of Satisfiability Testing* (pp. 299–314). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/978-3-540-24605-3_23`.

Boussemart, F., Lecoutre, C., & Piette, C. (2017). XCSP3: an integrated format for benchmarking combinatorial constrained problems. *The Computing Research Repository*, *abs/1611.03398*, 1–235. URL: `http://arxiv.org/abs/1611.03398`.

Correas, J., Martín, S. E., & Sáenz-Pérez, F. (2018). Enhancing set constraint solvers with bound consistency. *Expert Systems with Applications*, *92*, 485–494. doi:`10.1016/j.eswa.2017.09.056`.

Eén, N., & Biere, A. (2005). Effective Preprocessing in SAT Through Variable and Clause Elimination. In F. Bacchus, & T. Walsh (Eds.), *Theory and Applications of Satisfiability Testing* (pp. 61–75). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/11499107_5`.

Eén, N., & Sörensson, N. (2004). An Extensible SAT-solver. In E. Giunchiglia, & A. Tacchella (Eds.), *Theory and Applications of Satisfiability Testing* (pp. 502–518). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:`10.1007/978-3-540-24605-3_37`.

Frühwirth, T. (2009). *Constraint Handling Rules*. (1st ed.). Cambridge, UK: Cambridge University Press.

Ftulis, S., Giordano, M., Plüss, J., & Vota, R. (1998). Rule-based constraints programming: application to crew assignment. *Expert Systems with Applications*, *15*, 77–85. doi:`10.1016/S0957-4174(98)00013-X`.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability, A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman & Company.

Gent, I. P., & Lynce, I. (2005). A sat encoding for the social golfer problem. In *IJCAI'05 workshop on modelling and solving problems with constraints*.

Gervet, C. (1994). Conjunto: Constraint Propagation over Set Constraints with Finite Set Domain Variables. In *Proceedings of the Eleventh International Conference on Logic Programming* (p. 733). Cambridge, MA, USA: MIT Press. URL: `http://dl.acm.org/citation.cfm?id=189883.189954`.

Hamiez, J., & Hao, J. (2014). A note on a sports league scheduling problem. *The Computing Research Repository*, *abs/1410.2721*. URL: `http://arxiv.org/abs/1410.2721`.

Harvey, W. (2019). CSPLib problem 010: Social golfers problem. `http://www.csplib.org/Problems/prob010`.

Hawkins, P., Lagoon, V., & Stuckey, P. J. (2005). Solving Set Constraint Satisfaction Problems using ROBDDs. *Journal of Artificial Intelligence Research*, *24*, 109–156.

Hsiao, M. Y., Bossen, D. C., & Chien, R. T. (1970). Orthogonal latin square codes. *IBM Journal of Research and Development*, *14*, 390–394. doi:10.1147/rd.144.0390.

Lardeux, F., & Monfroy, E. (2014). From declarative set constraint models to "good" sat instances. In G. A. Aranda-Corral, J. Calmet, & F. J. Martín-Mateos (Eds.), *Artificial Intelligence and Symbolic Computation* (pp. 76–87). Cham: Springer International Publishing. doi:10.1007/978-3-319-13770-4_8.

Lardeux, F., & Monfroy, E. (2016). From Set Constraint Models to SAT Instances. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)* (pp. 231–238). doi:10.1109/ICTAI.2016.0044.

Lardeux, F., Monfroy, E., Crawford, B., & Soto, R. (2015). Set constraint model and automated encoding into SAT: application to the social golfer problem. *Annals OR*, *235*, 423–452. doi:10.1007/s10479-015-1914-5.

Lardeux, F., Monfroy, E., Saubion, F., Crawford, B., & Castro, C. (2009). SAT Encoding and CSP Reduction for Interconnected Alldiff Constraints. In A. H. Aguirre, R. M. Borja, & C. A. R. Garciá (Eds.), *MICAI 2009: Advances in Artificial Intelligence* (pp. 360–371). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-05258-3_32.

Lee, J. K., & Kwon, S. B. (1995). ES∗: An expert systems development planner using a constraint and rule-based approach. *Expert Systems with Applications*, *9*, 3–14. doi:10.1016/0957-4174(94)00043-U.

Mackworth, A. (1992). Constraint satisfaction. In S. Shapiro (Ed.), *Encyclopedia on Artificial Intelligence* chapter Constraint Satisfaction. (pp. 285–293). John Wiley.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., & Tack, G. (2007). MiniZinc: Towards a Standard CP Modelling Language. In C. Bessière (Ed.), *Principles and Practice of Constraint Programming – CP 2007* (pp. 529–543). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-540-74970-7_38.

Nightingale, P., & Miguel, I. (2018). Savile Row. http://savilerow.cs.st-andrews.ac.uk/. URL: \url{http://savilerow.cs.st-andrews.ac.uk/}.

Pegg, E., Jr. (2007). *Social Golfer Problem*. Math Games, mathpuzzle.com. URL: http://www.mathpuzzle.com/MAA/54-Golf%20Tournaments/mathgames_08_14_07.html.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2017). *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. URL: http://www.choco-solver.org.

Rossi, F., van Beek, P., & Walsh, T. (Eds.) (2006). *Handbook of Constraint Programming*. (1st ed.). Elsevier Science.

Stinson, D. R. (1994). Universal hashing and authentication codes. *Designs, Codes and Cryptography*, *4*, 369–380. doi:10.1007/BF01388651.

Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., & Fischer, J. (2014). The MiniZinc challenge 2008-2013. *AI Magazine*, *35*, 55–60. doi:10.1609/aimag.v35i2.2539.

Topaloglu, S., Salum, L., & Supciller, A. A. (2012). Rule-based modeling and constraint programming based solution of the assembly line balancing problem. *Expert Systems with Applications*, *39*, 3484–3493. doi:10.1016/j.eswa.2011.09.038.

Triska, M., & Musliu, N. (2012). An improved SAT formulation for the social golfer problem. *Annals of Operations Research*, *194*, 427–438. doi:10.1007/s10479-010-0702-5.

Walsh, T. (2019). CSPLib problem 026: Sports tournament scheduling. http://www.csplib.org/Problems/prob026.

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*, 67–96. doi:10.1017/S1471068411000494.

Yip, J., & Van Hentenryck, P. (2011). Checking and filtering global set constraints. In J. Lee (Ed.), *Principles and Practice of Constraint Programming – CP 2011* (pp. 819–833). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-23786-7_61.