Constraints (2013) 18:236–268
DOI 10.1007/s10601-012-9131-1

Solving weighted CSPs with meta-constraints by reformulation into satisfiability modulo theories

Carlos Ansótegui · Miquel Bofill · Miquel Palahí · Josep Suy · Mateu Villaret

Published online: 13 November 2012

Abstract We introduce `WSimply`, a new framework for modelling and solving Weighted Constraint Satisfaction Problems (WCSP) using Satisfiability Modulo Theories (SMT) technology. In contrast to other well-known approaches designed for extensional representation of goods or no-goods, and with few declarative facilities, our approach aims to follow an intensional and declarative syntax style. In addition, our language has built-in support for some meta-constraints, such as *priority* and *homogeneity*, which allows the user to easily specify rich requirements on the desired solutions, such as preferences and fairness. We propose two alternative strategies for solving these WCSP instances using SMT. The first is the reformulation into Weighted SMT (WSMT) and the application of satisfiability test based algorithms from recent contributions in the Weighted Maximum Satisfiability field. The second one is the reformulation into an operation research-like style which involves an

This work is an extended version of the paper [1] presented at the ModRef 2011 workshop.
Research partially supported by the Generalitat de Catalunya under grant AGAUR 2009-SGR-1434, and the Ministerio de Economía y Competividad research projects TIN2008-04547, TIN2009-14704-C03-01, and TIN2010-20967-C04-01/03.

C. Ansótegui
Departament d'Informàtica i Enginyeria Industrial, Universitat de Lleida,
25001 Lleida, Spain
e-mail: carlos@diei.udl.cat

M. Bofill (✉) · M. Palahí · J. Suy · M. Villaret
Departament d'Informàtica i Matemàtica Aplicada, Universitat de Girona,
17071 Girona, Spain
e-mail: mbofill@ima.udg.edu

M. Palahí
e-mail: mpalahi@ima.udg.edu

J. Suy
e-mail: suy@ima.udg.edu

M. Villaret
e-mail: villaret@ima.udg.edu

optimisation variable or objective function and the application of optimisation SMT solvers. We present experimental results of two well-known problems: the Nurse Rostering Problem (NRP) and a variant of the Balanced Academic Curriculum Problem (BACP), and provide some insights into the impact of the addition of meta-constraints on the quality of the solutions and the solving time.

Keywords Weighted CSP · Modelling languages · Reformulation · Meta-constraints · SMT · Weighted MaxSAT

1 Introduction

A Constraint Satisfaction Problem (CSP) is a decision problem where the objective is to determine whether an assignment of values to a set of variables exists which satisfies a given set of constraints. However, many real-world instances of CSPs are over-constrained and therefore have no solution. In this case, we can relax the CSP specification so that a number of constraints can be violated, and ask for a solution that maximises the number of satisfied constraints. This optimisation variant is known as Maximum CSP (MaxCSP) [16, 20, 21].

In some problems there can be a degree of preference on which constraints to violate. This is usually modelled by attaching a weight to each constraint, which denotes the cost of violating it. We refer to the constraints that can be violated as *soft*, while we refer to those constraints that must be satisfied as *hard* (their associated weight is considered to be infinite). In this case, the objective is to find an assignment which satisfies all the hard constraints and minimises the aggregated cost of the violated soft constraints [22]. This problem is known as the Weighted CSP (WCSP) [25] or, alternatively, as a Cost Function Network (CFN) [11, 12]. In a CFN, constraints are replaced by cost functions. Cost functions associate a cost with each combination of values which a set of variables can take, and the objective is to find an assignment of values for the variables that minimises the sum of all costs.

We may wish to go further at the specification level by allowing the user to express their preferences more easily, or even to express more complex preferences. In [32], a set of constraints on soft constraints, called *meta-constraints*, is introduced. Meta-constraints can be very helpful in the modelling process, since they allow us to abstract to a higher level, expressing, e.g., priorities between a set of soft constraints, different levels of preference (multi-objective optimisation), etc. For example, in the well-known Nurse Rostering Problem (NRP) [9], it is preferable to violate the constraint about the number of consecutive days that a nurse can work than the constraint about the minimum number of nurses per shift. We could also wish to impose certain homogeneity on the amount of violation of different sets of constraints. Continuing with the example of the NRP, this could be useful in order to obtain "fair" solutions with respect to the preferences of the nurses. Additionally, we may wish to consider dependencies between constraints (for instance, if a soft constraint is violated then some other constraint must become mandatory) or we may wish to define the weight of a soft constraint not as a constant value but as a function denoting its degree of violation.

Several soft constraint frameworks have been proposed. XCSP 2.1 [33] is an XML format which has been adopted recently in the CSP, MaxCSP and WCSP competitions. CSP instances, as well as WCSP instances, can be represented either in extension or in intension with XCSP. WCSP instances are represented in intension by introducing cost functions. Unfortunately, up to now the competition has been restricted to extensional instances in the case of WCSP. For this reason, all available WCSP benchmarks written in XCSP are described in extension, and we are not aware of any WCSP solver supporting intensional cost functions at this moment.

Some tools supporting higher-level and less verbose languages, such as MiniZinc [28], can output to XCSP. There are also tools like TAILOR [18], which translate from XCSP to other declarative, solver-independent modelling languages such as ESSENCE' (a subset of ESSENCE [17]). However, to our knowledge, none of these higher-level languages includes direct support for WCSP or meta-constraints. Comet [27] is a constraint programming system based on local search, with support for weighted constraints. The weights in a Comet specification can be either static or dynamic (i.e., they can vary during execution) in order to guide the local search.

In this work we follow a purely declarative approach, i.e., we focus on the intensional specification of WCSPs in a declarative high-level language, and on its resolution by means of general purpose, off-the-shelf decision procedures. We introduce a new framework, called `WSimply`, which includes a new language for modelling WCSPs with built-in support for meta-constraints, covering all the kinds described in [32], and a solving system for WCSPs with Satisfiability Modulo Theories (SMT) technology. In particular, we extend the medium-level constraint modelling language of `Simply` [7] to deal with weights associated with the violation of constraints.

`Simply` is able to solve decision and optimisation CSP instances by reformulating them into SMT and calling an external SMT solver. There are existing works which show the efficiency of SMT when dealing with particular CSPs [2, 29] and also when dealing with CSPs in general [7, 8]. With the proposed `WSimply` extension, the user will be able to model WCSP instances with meta-constraints easily, and choose to solve them either by reformulation into SMT following an operation research-like style and using binary search optimisation or, alternatively, by reformulation into WSMT and using satisfiability test based algorithms from recent contributions in Weighted MaxSAT.

We conduct extensive experimental evaluation on two well-known problems, the aforementioned NRP and a variant of the Balanced Academic Curriculum Problem (BACP). We show how to model these problems with the help of meta-constraints and discuss the performance of different solving techniques.

The rest of the paper is structured as follows. In Section 2 we recall some basic concepts on WCSP. In Section 3 we introduce SMT and WSMT. Section 4 is devoted to our framework and its language. In Section 5 we give an example of an over-constrained problem modelled using our language. In Section 6 we discuss several ways of solving WCSP instances with SMT. Results of the experimental evaluation are given in Section 7. We conclude in Section 8.

2 Weighted CSP (WCSP)

Next we formally define CSP and WCSP.

Definition 1 A *Constraint Satisfaction Problem (CSP)* instance is defined as a triple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{d(x_1), \ldots, d(x_n)\}$ is a set of domains containing the values the variables may take, and $C = \{C_1, \ldots, C_m\}$ is a set of constraints. Each constraint $C_i = \langle S_i, R_i \rangle$ is defined as a relation R_i over a subset of variables $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$, called the *constraint scope*. A relation R_i may be represented *intensionally* in terms of an expression which defines the relationship that must hold amongst the assignments to the variables it constrains or it may be represented *extensionally* as a subset of the Cartesian product $d(x_{i_1}) \times \cdots \times d(x_{i_k})$ (tuples) which represents the allowed assignments (good tuples) or the disallowed assignments (no-good tuples).

An *assignment* v for a CSP instance $\langle X, D, C \rangle$ is a mapping that assigns to every variable $x_i \in X$ an element $v(x_i) \in d(x_i)$.

An assignment v satisfies a constraint $\langle \{x_{i_1}, \ldots, x_{i_k}\}, R_i \rangle$ in C if and only if $\langle v(x_{i_1}), \ldots, v(x_{i_k}) \rangle \in R_i$.

A solution to a CSP instance is an assignment that satisfies all the constraints. The Constraint Satisfaction Problem for a CSP instance consists of finding a solution for that instance.

Definition 2 A *weighted CSP (WCSP)* instance is a triple $\langle X, D, C \rangle$, where X and D are variables and domains, respectively, as in a CSP. A constraint C_i is now defined as a pair $\langle S_i, f_i \rangle$, where $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ is the constraint scope and $f_i : d(x_{i_1}) \times \cdots \times d(x_{i_k}) \rightarrow \mathbb{N} \cup \{\infty\}$ is a *cost (weight) function* that maps tuples to its associated weight (a natural number or infinity). We call those constraints whose associated cost is infinity *hard*, if otherwise *soft*. The cost (weight) of a constraint C_i induced by an assignment v in which the variables of $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ takes values $b_{i_1}, \ldots, b_{i_k}$ is $f_i(b_{i_1}, \ldots, b_{i_k})$.

A solution to a WCSP instance is an assignment in which the sum of the costs of the constraints is minimal. The Weighted Constraint Satisfaction Problem for a WCSP instance consists of finding a solution for that instance.

In the literature a WCSP is also referred to as a constraint optimisation problem which is a regular CSP whose constraints are weighted and the goal is to find a solution while minimising the cost of the constraints. Nevertheless, in this paper we refer to a Constraint Optimization Problem (COP) in a more operation research-like style. This allows us to stress the difference between instances explicitly containing soft constraints and instances just containing an optimisation variable.

Definition 3 A Constraint Optimisation Problem (COP) instance consists of an optimisation variable O to be minimised (or maximised) subject to the constraints of a CSP instance $\langle X, D, C \rangle$ where $O \in X$.

A solution to a COP instance is a solution to the CSP instance that minimises (or maximises) the value of the optimisation variable.

3 Weighted SMT (WSMT)

In recent decades SAT solvers have progressed spectacularly in performance thanks to better implementation techniques and conceptual enhancements, such as non-chronological backtracking and conflict-driven lemma learning, which in many instances of real problems are able to reduce the size of the search space significantly [6, 15, 23]. Thanks to those advances, nowadays the best SAT solvers can tackle problems with hundreds of thousands of variables and millions of clauses.

An SMT instance is a generalisation of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a formula can contain clauses like $p \vee q \vee (x + 2 \leq y) \vee (x > y + z)$, where p and q are Boolean variables and x, y and z are integer variables. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory [30, 34].

Leveraging the advances made in SAT solvers in the last decade, SMT solvers have proved to be competitive with classical decision methods in many areas. Most modern SMT solvers integrate a SAT solver with decision procedures (theory solvers) for sets of literals belonging to each theory. It is worth noting that most state-of-the-art SMT solvers use the simplex method for dealing with linear integer arithmetic predicates. This way, we can hopefully get the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms for the theory reasoning.

As in the CSP case, we can extend SMT to WSMT as follows:

Definition 4 A *weighted SMT clause* is a pair (C, w), where C is an SMT clause[1] and w is a natural number or infinity (indicating the penalty for violating C). A *weighted SMT formula* is a multiset of weighted SMT clauses

$$\varphi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$$

where the first m clauses are soft and the last m' clauses are hard.

The optimal cost of a formula is the minimal cost of all its assignments. An optimal assignment is an assignment with optimal cost.

Definition 5 The WSMT problem[2] for a WSMT formula is the problem of finding an optimal assignment for that formula.

The WSMT problem can be reformulated in a operation research-like style, i.e., the maximisation of a linear function subject to a set of constraints, as follows:

$$\text{maximise } o_1 \cdot w_1 + \cdots + o_m \cdot w_m \tag{1}$$

subject to:

$$\bigwedge_{i=1}^{m} o_i \leftrightarrow C_i \tag{2}$$

$$\bigwedge_{j=m+1}^{m+m'} C_j \tag{3}$$

[1]In fact these can be general SMT formulas, not necessarily disjunctions of literals.

[2]In the literature the weighted SMT problem is also referred to as Weighted MaxSMT, same as in the SAT formalism. We prefer to use WSMT because it is closer to WCSP.

where o_i are Boolean variables, (1) is the linear function to be maximised, (3) is the original set of hard constraints and (2) ensures that o_i is true iff C_i is evaluated to true. In order to convert this maximisation problem into an equivalent minimisation problem we just need to replace (1) by:

$$\text{minimize} \sum_{i=1}^{m} w_i - (o_1 \cdot w_1 + \cdots + o_m \cdot w_m) \tag{4}$$

It can be easily seen that the implication $C_i \to o_i$ is unnecessary (as we are optimising).

4 Weighted Simply (WSimply)

`Simply`[3] [7] was developed as a declarative programming system for easy modelling and solving of CSPs. `Simply` essentially translates a CSP instance intensionally described in its own language into an SMT instance, which is fed into an SMT solver. If the SMT solver is able to find a solution, then this solution is translated back to a solution of the original CSP instance. Moreover, to deal with COP instances (Definition 3) optimisation algorithms can be run on top of an SMT solver [31]. Currently, `Simply` is integrated with the Yices SMT solver [14], using its built-in API. However, it could be easily adapted to work with other SMT solvers, either by using external files or any other API.

The language of `Simply` is similar to other high-level languages such as those of ESSENCE and MiniZinc. The model and the data can be specified in separate files, and the decision variables can be either Boolean or finite domain integer variables. The language has the useful declarative facility of list comprehensions, which allows for concise and elegant modellings.

Example 1 shows how to model an instance of the NRP with `Simply`, and the resulting SMT formula. As we can see, the SMT formula has no solution due to the preferences of the nurses which we have defined. The typical approach is to declare these preferences as soft constraints.

Example 1 Consider a simple instance of the NRP (details on the NRP are given in Section 5) with two shifts per day and two available nurses. Each shift must be covered with exactly one nurse. We also want to satisfy the preferences of the nurses. Say nurse 1 wants to work both days on shift 1, while nurse 2 wants to work the first day on shift 2 and the second day on shift 1.

This particular instance can be modelled with `Simply` as shown in Fig. 1. We can identify four sections in the `Simply` model: data, domains, variables and constraints. The data section allows us to define a particular instance of the problem. This section can also defined in a separate file. The variables section declares a bidimensional array of integer variables `nd`, where the size of the first dimension corresponds to the number of nurses, and the size of the second dimension to the number of days. The domain of the integer variables of this array is restricted to the range specified by `dshifts_types`, i.e., the possible shifts. In the constraints section, we first

[3] `Simply` and `WSimply` are available at http://ima.udg.edu/recerca/lap/simply/.

Fig. 1 A `Simply` instance

```
Problem:nrp
   Data
         int nurses = 2; int days = 2; int shifts = 2;
   Domains
         Dom dshifts_types = [1..shifts];
   Variables
         IntVar nd[nurses, days] :: dshifts_types;
   Constraints
         % Covers
         Forall(d in [1..days], st in [1..shifts]) {
            Count([nd[n, d] | n in [1..nurses]], st, 1);
         };
         % Preferences
         nd[1,1] = 1; nd[1,2] = 1; nd[2,1] = 2; nd[2,2] = 1;
```

introduce the cover constraints: for each day and shift, we have to meet the cover requirements (one nurse in this case). To post this constraint we generate the list of nurses working every day `d` and shift `st` with a list comprehension `[nd[n, d] | n in [1..nurses]]`, and we use the global constraint `Count` to restrict the number of nurses working that day and shift to one. Finally, we add the nurse preference constraints for each day, specifying which shift they prefer.

`Simply` translates the previous instance into the standard SMT-LIB v2 language [5] as shown in Fig. 2. First of all we need to specify the background theory to be used. In this example we use Linear Integer Arithmetic (LIA). Then, we define the variables of the problem as integer. Next, we use the `assert` operator to post the constraints, which are described in prefix notation. The first set of constraints bounds the domain of the variables. The second set corresponds to the translation

Fig. 2 SMT instance obtained from the `Simply` instance in Fig. 1

```
; The logic to be used
(set-logic QF_LIA)

; Variable declarations
(declare-fun nd_1_1 () Int)
(declare-fun nd_1_2 () Int)
(declare-fun nd_2_1 () Int)
(declare-fun nd_2_2 () Int)

; Bounds on the domain of the variables
(assert (and (<= nd_1_1 2) (>= nd_1_1 1)))
(assert (and (<= nd_1_2 2) (>= nd_1_2 1)))
(assert (and (<= nd_2_1 2) (>= nd_2_1 1)))
(assert (and (<= nd_2_2 2) (>= nd_2_2 1)))

; Cover constraints
(assert
  (and
        (= (+ (ite (= nd_1_1 1) 1 0) (ite (= nd_2_1 1) 1 0)) 1)
        (= (+ (ite (= nd_1_1 2) 1 0) (ite (= nd_2_1 2) 1 0)) 1)
        (= (+ (ite (= nd_1_2 1) 1 0) (ite (= nd_2_2 1) 1 0)) 1)
        (= (+ (ite (= nd_1_2 2) 1 0) (ite (= nd_2_2 2) 1 0)) 1)
  )
)

; Preference constraints
(assert (= nd_1_1 1))
(assert (= nd_1_2 1))
(assert (= nd_2_1 2))
(assert (= nd_2_2 1))

(check-sat)
```

of the cover constraints. Here, we use the operator `ite` (if-then-else, or conditional expression). For example, the expression `(ite (= nd_1_1 1) 1 0)` evaluates to 1 when `nd_1_1` is equal to 1, and to 0 otherwise. Each of the four lines with the `ite` operator guarantees that there is exactly one nurse per shift. Finally, we add the translation of the preference constraints.

4.1 Soft constraints in WSimply

Here we detail the extensions introduced to `Simply` in order to allow for soft constraints. The new language and system is called `WSimply`. The basic type of soft constraint in `WSimply` is of the form:

$$(\textit{constraint})\ @\ \{\textit{expression}\};$$

where the value of *expression* is the weight (cost) of falsifying the associated *constraint*. The parentheses around the constraint and the curly brackets around the weight expression are optional. The expression must be a linear integer arithmetic expression.[4] It can either be evaluable at compile time, or contain decision variables. Weight expressions should always evaluate to a non-negative integer, as they amount to a cost. When negative they will be considered as zero.

Example 2 The instance in Example 1 has no solution. Hence, we could wish to model the preferences as soft constraints (and give, e.g., double importance to the preferences of nurse 1). This could be modelled in `WSimply` by replacing the preference constraints with the following:

```
 % Preferences
(nd[1,1] = 1) @ 2;    (nd[1,2] = 1) @ 2;
(nd[2,1] = 2) @ 1;    (nd[2,2] = 1) @ 1;
```

Most (if not all) existing WCSP solving systems consider an extensional approach, i.e., they deal with instances consisting of an enumeration of good/no-good tuples for hard constraints, and no-good tuples with an associated cost for soft constraints.

Our proposal follows the other direction and aims to allow the user to model soft constraints in intension. Consider, for instance, two variables x and y, with domain $\{1, 2\}$, and the soft constraint $x < y$ with falsification cost 1. In an extensional approach, we would model this problem with the following soft no-good tuples: $(x = 1, y = 1, 1)$, $(x = 2, y = 1, 1)$ and $(x = 2, y = 2, 1)$. In `WSimply` we would express this with the soft constraint `(x < y) @ {1};`.

4.1.1 Degree of violation

An interesting detail worth remarking on is that by allowing the use of decision variables in the cost expression we can encode the degree of violation of a constraint.

[4]We are restricted to linear expressions, since we rely on SMT solvers and only a few of them incorporate (some limited) support for non-linear expressions.

For instance, for a nurse working more than five shifts in a week, the violation cost could be increased by one unit for each extra shift worked:

```
(worked_shifts < 6) @ {base_cost + worked_shifts − 5};
```

4.1.2 Labelled constraints

Soft constraints can be labelled as follows (again, the parentheses around the constraint and the curly brackets around the weight expression are optional):

$$\#label : (constraint) \ @ \ \{expression\};$$

The labels can be used to refer to the respective constraints in other (either soft or hard) constraints. For example:

```
#A: (a > b) @ 1;
#B: (a > c) @ 2;
#C: (a > d) @ 1;
(Not A And Not B) Implies C;
```

Labels are also used in meta-constraints, which we introduce in the next section. Moreover, indexed labels are supported, as we show in Section 5.2. This allows for convenient modellings in many cases.

4.2 Meta-constraints in WSimply

In order to provide a higher level of abstraction in the modelling of over-constrained problems, in [32] several meta-constraints are proposed. By meta-constraint we refer to a constraint on constraints. Meta-constraints allow us to go one step further in the specification of the preferences on the soft constraint violations. `WSimply` covers all meta-constraints introduced in [32], plus several variants and alternative meta-constraints, that we have grouped in the following three families:

1. **Priority.** The user may have some preferences about which soft constraints to violate. For instance, if there is an activity to perform and worker 1 *does not want* to perform it while worker 2 *should not* perform it, then it is better to violate the first constraint than the second. It would be useful to free the user of deciding the exact value of the weight of each constraint. To this end, we allow the use of undefined weights, denoted by "_":

$$\#label : (constraint) \ @ \ \{\_\};$$

   The value of this undefined weight is computed at compile time according to the priority meta-constraints that refer to the label. This simplifies the modelling of the problem, since the user does not need to compute any concrete weight. `WSimply` provides the following meta-constraints related to priority:

   – `samePriority(`*List*`)`, where *List* is a list of labels of soft constraints. This meta-constraint gives the same priority, i.e., the same weight, to the constraints denoted by the labels in *List*.
   – `priority(`*List*`)`, where *List* is a list of labels of soft constraints. This constraint orders the constraints denoted by the labels in *List* by decreasing priority. In other words, it imposes decreasing weights.

- `priority(`$label_1$`,`$label_2$`,`n`)`, with $n > 1$, defines how many times it is worse to violate the constraint corresponding to $label_1$ than to violate the constraint corresponding to $label_2$. That is, if $weight_1$ and $weight_2$ denote the weights associated with $label_1$ and $label_2$ respectively, it states $weight_1 \geq weight_2 * n$.
- `multiLevel(`*ListOfLists*`)`, where *ListOfLists* is a list of lists of labels. This meta-constraint states that the weight of each one of the constraints (denoted by the labels) in each list is greater than the aggregated weight of the constraints in the following lists. For example,

  ```
  multiLevel([[A,B,C],[D,E,F],[G,H,I]]);
  ```

  states that the cost of falsifying each of the constraints (denoted by) `A`, `B` and `C` is greater than the cost of falsifying `D`, `E`, `F`, `G`, `H`, and `I` together and, at the same time, the cost of falsifying each of the constraints `D`, `E` and `F` is greater than the cost of falsifying `G`, `H`, and `I` together.

2. **Homogeneity.** The user may wish there to be some homogeneity in the amount of violation of disjoint groups of constraints. For instance, for the sake of fairness, the number of violated preferences of nurses should be as homogeneous as possible.
   `WSimply` provides the following meta-constraints related to homogeneity:

   - `atLeast(`*List*`,`p`)`, where *List* is a list of labels of soft constraints and p is a positive integer in 1..100. This meta-constraint ensures that the percentage of constraints denoted by the labels in *List* that are satisfied is at least p.
   - `homogeneousAbsoluteWeight(`*ListOfLists*`,`v`)`, where *ListOfLists* is a list of lists of labels of soft constraints and v is a positive integer. This meta-constraint ensures that, for each pair of lists in *ListOfLists*, the difference between the cost of the violated constraints in the two lists is at most v. For example, given

     ```
     homogeneousAbsoluteWeight([[A,B,C],[D,E,F,G]],10);
     ```

     if the weights of constraints `A`, `B` and `C` are 5, 10 and 15 respectively, and constraints `A` and `B` are violated and constraint `C` is satisfied, then the cost of the violated constraints in `[D,E,F,G]` must be between 5 and 25.
   - `homogeneousAbsoluteNumber(`*ListOfLists*`,`v`)`. Same as above, but where the maximum difference v is between the number of violated constraints.
   - `homogeneousPercentWeight(`*ListOfLists*`,`p`)`, where *ListOfLists* is a list of lists of labels of soft constraints and p is a positive integer in 1..100. This meta-constraint is analogous to `homogeneousAbsoluteWeight`, but where the maximum difference p is between the percentage in the cost of the violated constraints (with respect to the cost of all the constraints) in each list.
   - `homogeneousPercentNumber(`*ListOfLists*`,`p`)`. Same as above, but where the maximum difference p is between the percentage in the number of violated constraints.

We remark that the `homogeneousAbsoluteWeight` and `homogeneous-PercentWeight` meta-constraints are not allowed to refer to any constraint with undefined weight. This is because, as aforementioned, constraints with

undefined weight are referenced by priority meta-constraints, and their weight is determined at compile time accordingly to those priority meta-constraints, independently from other constraints.
Hence, since the `homogeneousAbsoluteWeight` and `homogeneousPercentWeight` meta-constraints also constrain the weight of the referenced constraints, if they were allowed to reference constraints with undefined weights, this could lead to incompleteness. We also remark that these two meta-constraints cannot refer to constraints whose weights contain decision variables (see Section 6.1.2). Nevertheless, note that we can homogenise constraint violations using `homogeneousAbsoluteNumber` and `homogeneousPercentNumber` meta-constraints, without worrying about the weights.

3. **Dependence.** Particular configurations of violations may entail the necessity of satisfying other constraints, that is, if a soft constraint is violated then another soft constraint must not be violated, or a new constraint must be satisfied. For instance, in the context of the NRP, we can imagine that working the first or the last shift of the day is penalised and, if somebody works in the last shift of one day, then they cannot work in the first shift the next day. This could be succinctly stated as follows:

```
#A: not_last_shift_day_1 @ w1;
#B: not_first_shift_day_2 @ w2;
(Not A) Implies B;
```

stating that, if constraint `A` is violated, then constraint `B` becomes mandatory.

Although the priority meta-constraints are discussed in [32], they are not really developed in detail, and the multilevel meta-constraint is not considered. Our `atLeast` homogeneity meta-constraint subsumes the homogeneity meta-constraint defined in [32], while the `homogeneousAbsoluteWeight`, `homogeneous-homogeneousAbsoluteNumber`, `homogeneousPercentWeight` and `homogeneousPercentNumber` meta-constraints are new. The dependence meta-constraints are the same as in [32].

4.3 Related work

XCSP [33] is another WCSP specification language that accepts weights for intensional constraints. In particular, cost functions can be described intensionally. Although cost functions (see Definition 2) described intensionally in XCSP may seem more general than what `WSimply` can allow, we can easily emulate them. Given a cost function f, a naive translation in `WSimply` corresponds to the soft constraint *False* @ w, where w is the translation of f into the `Simply` language. Since the soft constraint is trivially false, we always add the cost represented by w. Notice that cost functions from Definition 2 range from 0 to ∞, where 0 corresponds to a soft constraint that is satisfied while ∞ corresponds to a hard constraint that is falsified. However, as we have discussed earlier, w is restricted to being a linear integer arithmetic expression since the current `WSimply` SMT-based solving methods only use the LIA theory. This can be circumvented by extending our SMT-based solving methods with additional theories or including other solving approaches with richer input languages.

There is a subtle detail in XCSP which we currently do not address in WSimply: the explicit definition of the top weight. We implicitly assume the top weight is infinite since we work with hard clauses. On the other hand, WSimply incorporates a set of meta-constraints and undefined weights, while XCSP does not.

Finally, XCSP is XML-based and therefore the description of the constraints is more bizarre than in WSimply. In WSimply, we provide list comprehensions and iterative constructs.

5 Modelling example

In this section we illustrate the use of WSimply meta-constraints on a paradigmatic example of over-constrained CSP: the Nurse Rostering Problem (NRP). In a NRP we have to generate a roster assigning shifts to nurses over a period of time subject to a number of constraints. These constraints, which can be hard or soft, are usually defined by regulations, working practices and nurse preferences [26].

We have chosen a simplified variant of the GPost NRP instance.[5] In this example we consider 4 weeks (28 days), 8 nurses (4 full-timers and 4 part-timers) and two shift types (day and night). We define an array variable sh[8,28] with domain [0..2], where sh[i,d] = 0 means that the i-th nurse does not work on day d, sh[i,d] = 1 means that the nurse works on the day shift of day d, and sh[i,d] = 2 means that the nurse works on the night shift of day d. Nurses numbered from 1 to 4 are full-timers, and from 5 to 8 are part-timers.

5.1 Hard constraints

Full-timers work exactly 18 shifts in 4 weeks, while part-timers work only 10. We can encode this as follows:

```
Forall(i in [1..8]) {
  If (i < 5) Then { Count( [sh[i,d] > 0 | d in [1..28]],
       True, 18 ); }
  Else            { Count( [sh[i,d] > 0 | d in [1..28]],
       True, 10 ); };
};
```

Each nurse works at most 4 night shifts, of which at most 3 are consecutive. In order to refer to the number of night shifts per worker we have to introduce another array variable tns[8] with domain [0..4]:

```
Forall(i in [1..8]) {
  Count( [sh[i,d] | d in [1..28]], 2, tns[i] );
};
% restrict the number of consecutive night shifts
Forall(i in [1..8], d in [1..25]) {
    ((sh[i,d] > 1) And (sh[i,d+1] > 1) And (sh[i,d+2] > 1))
    Implies Not(sh[i,d+3] > 1);
};
```

[5]http://www.cs.nott.ac.uk/~tec/NRP/

Note that the maximum number of night shifts is bounded by the domain of the array `tns`. Moreover, since the domain of the integers in array `sh` is `[0..2]`, we could alternatively write, e.g., `sh[i,d] = 2` instead of `sh[i,d] > 1`. However, we have observed that using strict inequalities often results in better performance, possibly due to the special treatment given to them by the linear integer arithmetic solver integrated with Yices [13].[6]

5.2 Soft constraints

There is a penalty for a single night shift (for the sake of simplicity, we ignore the first and last days of the roster):

```
Forall(i in [1..8], d in [1..26]) {
 #NSP[i,d]: Not((sh[i,d]<2) And (sh[i,d+1]>1) And
       (sh[i,d+2]<2)) @ {_};
};
```

Note that we can introduce arrays of labels in the `Forall` statement, which are indexed according to the `Forall` variables. We leave the weights undefined, since we just want all of them to be the same. To this end, we only need to post the following meta-constraint (which uses the indexed labels introduced in the `Forall` statement):

```
samePriority([NSP[i,d] | i in [1..8], d in [1..26]]);
```

Similarly, we want to penalise isolated free days (again, the first and last days of the roster are ignored for the sake of simplicity):

```
Forall(i in [1..8], d in [1..26] {
 #AFD[i,d]: Not((sh[i,d]>0) And (sh[i,d+1]<1) And
       (sh[i,d+2]>0)) @ {_};
};
samePriority([AFD[i,d] | i in [1..8], d in [1..26]]);
```

Since we consider that violating the `AFD` constraints is 10 times preferable to violating the `NSP` constraints, we use the following meta-constraint:

```
priority(NSP[1,1], AFD[1,1], 10);
```

Note that it is enough to state this priority between the first constraints of each group, since all constraints of each group have the same priority.

A full-timer has to work 4 or 5 days per week. We want to consider the deviation from this number as the violation degree of the constraint. This can be expressed

[6]Note that `WSimply` is using the Yices SMT solver as its default core solving engine.

as follows: by introducing an array variable `tw[4,4]`, where `tw[i,j]` denotes the number of working days for nurse `i` on week `j`:

```
Forall(i in [1..4]) {
  Count( [sh[i,d] > 0 | d in [1..7]], True, tw[i,1] );
  Count( [sh[i,d] > 0 | d in [8..14]], True, tw[i,2] );
  ...
};
Forall(i in [1..4], w in [1..4]) {
  Not(tw[i,w] > 5) @ {tw[i,w] - 5};
  Not(tw[i,w] < 4) @ {4 - tw[i,w]};
};
```

Finally, we can use homogeneity meta-constraints in order to guarantee a minimum satisfaction on the free days assigned to each nurse. This can be achieved as follows: we first state, as a soft constraint of weight 1, each free day requested by each nurse being free. We assume that each nurse has asked for five preferred free days, which are stored in an input data array `free[8,5]`.

```
Forall(i in [1..8], f in [1..5]) {
  #PFD[i,f]: (sh[i,free[i,f]] < 1) @ 1;
};
```

Then, the following meta-constraints can be used in order to guarantee that, globally, 40 % of the preferences of the nurses are satisfied and, at the same time, to homogeneously satisfy the preferences among nurses (the difference in the percentage of violated preferences for the different nurses is no more than 50):

```
atLeast([PFD[i,f] | i in [1..8], f in [1..5]], 40);
homogeneousPercentNumber([[PFD[1,f] | f in [1..5]],
                          [PFD[2,f] | f in [1..5]], ...], 50);
```

Finally, we assert that it is ten times better not to have an isolated free day than to rest one of the preferred days. This is also a requirement of the GPost instance:

```
priority(AFD[1,1], PFD[1,1], 10);
```

It is worth noting that, at this point, the system is able to determine a concrete value for the undefined weights of the `AFD` and `NSP` constraints.

6 Solving process

Figure 3 shows the basic architecture and solving process of `WSimply`. `WSimply` reformulates the input instance into the suitable format for the solving procedures. We have four reformulations: (R1) from a WCSP instance with meta-constraints into a WCSP instance (without meta-constraints), (R2) from a WCSP instance into a COP instance, (R3) from a WCSP instance into a WSMT instance and (R4) from a COP instance into a WSMT instance. Let us recall that the constraints in the WCSP and COP instances are expressed in the `WSimply` language as described in Section 4.

Once the problem has been properly reformulated, we can apply two different solving approaches: WSMT Solving (S1) or Optimisation SMT Solving (S2).

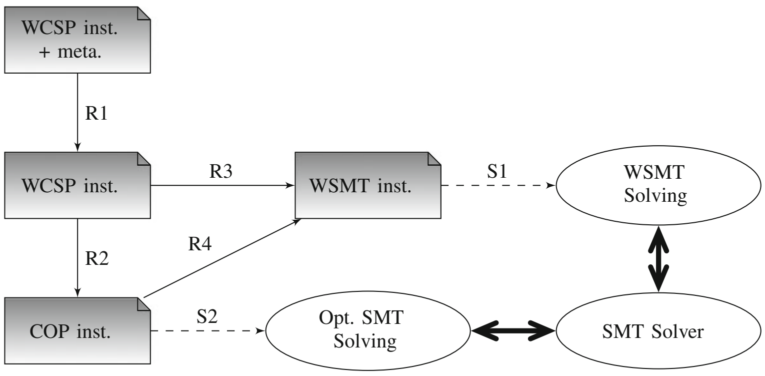

Fig. 3 Basic architecture and solving process of `WSimply`

In the following sections we describe the different reformulations and solving procedures.

6.1 Reformulating WCSP with meta-constraints into WCSP (R1)

We remove all the meta-constraints by reformulating them into hard and soft `WSimply` constraints.

As we have shown in Section 4, meta-constraints use labels. Therefore, first of all, we introduce a new reification variable for each labelled soft constraint of the form:

$$\#label : (constraint)\ @\ \{expression\};$$

and we replace the constraint by:

$$b_{label} \Leftrightarrow constraint;$$
$$b_{label}\ @\ \{expression\};$$

where b_{label} is the fresh (Boolean) reification variable introduced for this constraint.

In the following we show how `WSimply` reformulates the priority, homogeneity and dependence meta-constraints.

6.1.1 Reformulation of priority meta-constraints

To deal with the *priority* meta-constraints, we create a system of linear inequations on the (probably undefined) weights of the referenced soft constraints. The inequations are of the form $w = w'$, $w > w'$ or $w \geq n \cdot w'$, where w is a variable, w' is either a variable or a non-negative integer constant, and n is a positive integer constant. For example, given

```
    #A:(a>b)@{3};   #B:(a>c)@{_};   #C:(a>d)@{_};
    #D:(c=2-x)@{_};
    priority([A,B,C]);
    priority(D,B,2);
```

the following set of inequations is generated:

$$\begin{aligned}&w_A = 3, w_B > 0, w_C > 0, w_D > 0,\\&w_A > w_B, w_B > w_C,\\&w_D \geq 2 \cdot w_B\end{aligned}$$

This set of inequations is fed into an SMT solver[7] which acts as an oracle at compile time, so that a model, i.e., a value for the undefined weights satisfying the inequations, can be found. Following the previous example, the SMT solver would return a model such as, e.g.:

$$w_A = 3, w_B = 2, w_C = 1, w_D = 4$$

This allows the reformulation of the original problem into an equivalent WCSP without undefined weights:

```
    #A:(a>b)@{3};   #B:(a>c)@{2};   #C:(a>d)@{1};
    #D:(c=2-x)@{4};
```

Hence, with the meta-language, and thanks to this simple use of a solver as an oracle at compile time, we free the user of the tedious task of thinking about concrete weights for encoding priorities.

Since satisfiability test based algorithms (see Section 6.5) usually behave better with low weights, we ask not only for a solution of the inequations system, but for a solution minimising the sum of undefined weights.

In the case of the `multiLevel` meta-constraint, given for example

```
#A:(a>b)@{_};   #B:(a>c)@{_};   #C:(a>d)@{_};   #D:(c=2-x)@{_};
multiLevel([[A,B][C,D]]);
```

the following set of inequations would be generated:

$$\begin{aligned}&w_A > 0, w_B > 0, w_C > 0, w_D > 0,\\&w_A > (w_C + w_D),\\&w_B > (w_C + w_D),\end{aligned}$$

and the SMT solver would return a model such as, e.g.:

$$w_A = 3, w_B = 3, w_C = 1, w_D = 1$$

We remark that the weight expressions of the constraints referenced by a priority meta-constraint must either be undefined or evaluable at compile time, i.e., they cannot use any decision variable, since our aim is to compute all undefined weights at compile time. Moreover, if the set of inequations turns out to be unsatisfiable, the user will be warned about this fact during compilation.

[7] In fact, the set of inequations could be fed into any linear integer arithmetic solver.

6.1.2 Reformulation of homogeneity meta-constraints

We reformulate the *homogeneity* meta-constraints by reifying the referenced constraints and constraining the number of satisfied constraints. For example, the meta-constraint `atLeast(`*List*`,` *p*`)` is reformulated into:

```
Count(ListReif, True, n);
n >= val;
```

where `ListReif` is the list of Boolean variables resulting from reifying the constraints referenced in *List*, and `val` is computed in compile time and is equal to $\lceil length(List) * p/100\rceil$. `Count(`$l, e, n$`)` is a `Simply` global constraint that is satisfied if and only if there are exactly *n* occurrences of the element *e* in the list *l*.

The meta-constraint `homogeneousPercentWeight(`*ListOfLists*`,` *p*`)`, is reformulated into:

```
Sum([ weight_label[1][j] | j in [1..len[1] ],
  total_wei[1]);
Sum([ If_Then_Else (ListOfLists[1][j]) (0)
   (weight_label[1][j])
      | j in [1..len[1]] ], vio_wei[1]);
(vio_wei[1] * 100 Div total_wei[1]) >= min_homogen;
(vio_wei[1] * 100 Div total_wei[1]) =< max_homogen;

...

Sum([ weight_label[n][j] | j in [1..len[n] ],
  total_wei[n]);
Sum([ If_Then_Else (ListOfLists[n][j]) (0)
   (weight_label[n][j])
      | j in [1..len[n]] ], vio_wei[n]);
(vio_wei[n] * 100) Div total_wei[n]) >= min_homogen;
(vio_wei[n] * 100) Div total_wei[n]) =< max_homogen;
(max_homogen - min_homogen) < p;
```

where `len[i]` is the length of the `i`-th list in *ListOfLists*, `weight_label[i][j]` is the weight associated with the `j`-th label of the `i`-th list, `total_wei[i]` is the aggregated weight of the labels in the `i`-th list and `n` is the length of *ListOfLists*. Note that according to the `Sum` constraints, `vio_wei[i]` denotes the aggregated weight of the violated constraints in the `i`-th list. Finally, `min_homogen` and `max_homogen` are fresh new variables.

Since we are restricted to linear integer arithmetic, the `total_wei[i]` expressions must be evaluable at compile time. This requires the weights of the constraints referenced by this meta-constraint to be evaluable at compile time.

The reformulation of `homogeneousAbsoluteWeight(`*ListOfLists*`,` *v*`)` is analogous to the previous one, but where, instead of computing the percentage on `vio_wei[i]`, we can directly state:

```
...
vio_wei[1] >= min_homogen;
vio_wei[1] =< max_homogen;
```

```
...
vio_wei[n] >= min_homogen;
vio_wei[n] =< max_homogen;
(max_homogen - min_homogen) =< v;
```

Technically, our reformulation could allow the `homogeneousAbsoluteWeight` meta-constraint to reference constraints whose weight expression uses decision variables (and hence is not evaluable at compile time) without falling out of linear integer arithmetic. However, this is not the case for `homogeneousAbsolutePercent`, for which we must be able to compute the aggregated weight of the labels. Thus, for coherence reasons, we have forbidden both constraints to reference constraints with decision variables in their weight, in addition to constraints with undefined ("_") weight, as pointed out in Section 4.2.

The reformulations of `homogeneousAbsoluteNumber` and `homogeneousPercentNumber` are similar to the previous ones, but where we count the number of violated constraints instead of summing their weights.

6.1.3 Reformulation of dependence meta-constraints

The *dependence* meta-constraints are straightforwardly reformulated by applying the logical operators between constraints directly supported by `Simply` on the corresponding reification variables.

6.2 Reformulating WCSP into COP (R2)

In order to convert our WCSP instance into a COP instance, we first replace each soft constraint $C_i @ w_i$ with the following constraints where we introduce a fresh integer variable o_i:

$$\neg(w_i > 0 \wedge \neg C_i) \rightarrow o_i = 0 \tag{5}$$

$$(w_i > 0 \wedge \neg C_i) \rightarrow o_i = w_i \tag{6}$$

If the weight expression, w_i, evaluates to a value less or equal than 0, then the cost of falsifying C_i is 0, otherwise it is w_i. Since we are defining a minimisation problem we could actually replace (5) with $o_i \geq 0$.

Secondly, we introduce another fresh integer variable O, which represents the sum of the o_i variables, i.e., the optimisation variable of the COP to be minimised, and the following constraint:

$$O = \sum_{i=1}^{m} o_i \tag{7}$$

Finally, we keep the original hard constraints with no modification.

6.3 Reformulating WCSP into WSMT (R3)

To the best of our knowledge, existing WSMT solvers only accept WSMT clauses whose weights are constants. Therefore, we need to convert the WCSP instance into a WSMT instance where all the WSMT clauses have a constant weight.

We apply the same strategy as in R2, i.e., we first replace each soft constraint $C_i @ w_i$, where w_i is not a constant (involves variables), with the constraints (5) and (6) introducing a fresh integer variable o_i. Secondly, we add the following set of soft constraints over each possible value of each o_i variable:

$$\bigcup_{v_j \in V(o_i)} o_i \neq v_j \,@\, v_j \tag{8}$$

where $V(o_i)$ is the set of all possible positive values of o_i. These values are determined by evaluating the expression for all the possible values of the variables, and keeping only the positive results. Note that at this point we do have a WCSP instance where all the soft constraints have a constant weight.

Finally, we replace each soft constraint $C_i \,@\, w_i$, with the WSMT clause (C'_i, w_i) where C'_i is the translation of C_i into SMT as described in [7]. We also replace each hard constraint with its equivalent hard SMT clause.

6.4 Reformulating COP into WSMT (R4)

Taking into account that the optimisation variable O of the COP instance is the integer variable that represents the objective function, we only need to add the following set of WSMT clauses: $\bigcup_{i=1}^{i=W}(O < i, 1)$, where W is the greatest value the objective variable can be evaluated to. A more concise alternative could result from using the binary representation of W, i.e., adding the set of WSMT clauses $\bigcup_{i=0}^{i<\lceil \log_2(W+1) \rceil}(\neg b_i, 2^i)$, and the hard clause $(\sum_{i=0}^{i<\lceil \log_2(W+1) \rceil} 2^i \cdot b_i = O, \infty)$.

We finally replace all the constraints of the COP instance with the equivalent hard SMT clauses as described in [7].

6.5 Solving with SMT

From Fig. 3 we see that we can currently apply two solving methods in `WSimply`: WSMT solving which receives as input a WSMT instance and Optimisation SMT solving which receives as input a COP instance.

6.5.1 WSMT solving (S1)

The Yices SMT solver [14] offers a non-exact algorithm[8] to solve WSMT instances. We refer to this solving method as yices. Since this is still an immature research topic in SMT, we have extended the Yices framework by incorporating other exact algorithms from the MaxSAT field. There, we can find two main classes of algorithms: branch and bound based and satisfiability test based algorithms. The solvers that implement the latter clearly outperform branch and bound based solvers on industrial, and some crafted instances, and constitute an emerging technology.

In the following we describe the basic scheme of satisfiability test based algorithms. A WSMT problem φ can be solved through the resolution of a sequence of SMT instances as follows. Let φ_k be an SMT formula that is satisfiable if, and only if, φ has an assignment with a cost smaller than or equal to k (k plays the role of the bound which we impose on the objective function). If the cost of the optimal assignment to φ is k_{opt}, then the SMT problems φ_k, for $k \geq k_{\text{opt}}$, are satisfiable, while

[8]Non-exact algorithms do not guarantee optimality.

for $k < k_{\text{opt}}$ are unsatisfiable. Note that k may range from 0 to $\sum_{i=1}^{m} w_i$ (the sum of the weights of the soft clauses). When the weights are expressions rather than constants, we estimate an upper-bound. This is done by evaluating the expression for all the possible values of the variables. The search for the value k_{opt} can be done following different strategies; searching from $k = 0$ to k_{opt} (increasing k while φ_k is unsatisfiable); from $k = \sum_{i=1}^{m} w_i$ to a value smaller than k_{opt} (decreasing k while φ_k is satisfiable); or alternating unsatisfiable and satisfiable φ_k until the algorithm converges to k_{opt} (for instance, using a binary search scheme). The key point to boosting the efficiency of these approaches is to know whether we can exploit any additional information from the execution of the SMT solver for the subsequent runs.

Since `WSimply` is designed to use SMT solvers as a black box, satisfiability test based algorithms can be easily integrated into `WSimply`.

In particular, we have implemented the WPM1 algorithm from [4, 24], which is based on the detection of unsatisfiable cores. These are satisfiability test based algorithms, where the parameter k ranges from 0 to k_{opt}. Then, for every UNSAT answer, they analyse the core of unsatisfiability of the formula returned by the SMT solver. This information is incorporated in the form of redundant clauses into the next call to the SMT solver which help to boost the propagation. In our experiments, we refer to the method which uses the WPM1 algorithm as core.

The pseudo-code of the WPM1 algorithm based on calls to an SMT solver is described in Algorithm 1. This algorithm is the weighted version of the FuMalik algorithm [4, 24] for partial MaxSAT. In those works, the underlying solver was a SAT solver. This is the first time SMT technology has been incorporated in the implementation of Algorithm 1.

In Algorithm 1, we iteratively call an SMT solver with a weighted working formula φ, but excluding the weights. This corresponds to line 4. The SMT solver will say whether the formula is satisfiable or not (variable *st*) and in case the formula is unsatisfiable, it will give an unsatisfiable core (φ_c). When the SMT solver returns an unsatisfiable core, we compute the minimum weight of the clauses of the core (w_{min} in the algorithm). Then, we transform the working formula by duplicating

Algorithm 1: The pseudo-code of the WPM1 algorithm.

```
   Input: $\varphi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$
1  if $SMT(\{C_i \mid w_i = \infty\}) = (UNSAT, \_)$ then return $(\infty, \emptyset)$ ;   /* Hard clauses are unsatisfiable */
2  $cost := 0$;                                                     /* Optimal */
3  while $true$ do
4      $(st, \varphi_c) := SMT(\{C_i \mid (C_i, w_i) \in \varphi\})$ ; /* Call to the SMT solver without weights */
5      if $st = SAT$ then return $(cost, \varphi)$;
6      $BV := \emptyset$;                                  /* Blocking variables of the core */
7      $w_{min} := \min\{w_i \mid C_i \in \varphi_c \wedge w_i \neq \infty\}$ ;              /* Minimum weight */
8      foreach $C_i \in \varphi_c$ do
9          if $w_i \neq \infty$ then
10             $b :=$ new_variable();
11             $\varphi := \varphi \setminus \{(C_i, w_i)\} \cup \{(C_i, w_i - w_{min}), \ (C_i \vee b, w_{min})\}$
               ;                                   /* Duplicate soft clauses of the core */
12             $BV := BV \cup \{b\}$
13     $\varphi := \varphi \cup \{(exactly\_one(\{b \mid b \in BV\}), \infty)\}$
       ;                                  /* Add cardinality constraint as hard clauses */
14     $cost := cost + w_{min}$
```

the clauses in the core. Then, in one of the copies we give to the clauses their original weight minus the minimum weight. On the other copy, we extend the clauses with the blocking variables (BV in the code) and we give them the minimum weight. Finally, we add the cardinality constraint on the blocking variables using the standard encoding of the *exactly_one* Boolean constraint. Note that we could assert this cardinality constraint using the Linear Integer Arithmetic theory, however, the Boolean encoding has shown a better performance in our experiments. We finally add $w_{\min}$ to the *cost*.

6.5.2 Optimisation SMT solving (S2)

This method is also based on calling an SMT solver incrementally. We first translate all the constraints of the COP instance into SMT clauses and send them to the SMT solver. Then, before every call we bound the domain of the optimisation variable O by adding the SMT clause $O \leq k$, where k is an integer constant. If the SMT solvers returns `unsat`, we replace the clause $O \leq k$ by $O > k$. The strategy we use to determine the next k is a binary search. In the following, we will refer to this solving method as dico.

7 Benchmarking

In order to show the usefulness of meta-constraints we have conducted several experiments on a set of instances of the *Nurse Rostering Problem* (NRP) and on a variant of the *Balanced Academic Curriculum Problem* (BACP).

7.1 Nurse Rostering Problem

There exist many formalisations of the NRP [9]. In our experiments we have considered the (complete) GPost instance (of which we have modelled a variant in Section 5) as well as many instances from the Nurse Scheduling Problem Library (NSPLib) [35], by conducting a precise study of the effects of the homogeneity meta-constraints on them.

7.1.1 GPost NRP

In Section 5 we have already presented how to model a variant of the GPost NRP with `WSimply` including several meta-constraints. `WSimply` shows a reasonably good performance when solving the original GPost instance, compared to the results of the same problem reported in [26]. The authors report 8 s (2.83 GHz Intel® Core™ 2 Duo) for finding the optimal solution (cost 3) with an ad hoc search with CPLEX over a previously computed enumeration of all possible schedules for each nurse. They also report 234 s (2.8 GHz Pentium IV) for finding a non-optimal solution (cost 8) with their generic local search method (VNS/LDS+CP) based on neighbourhoods plus an exploration of the search space with CP and soft global constraints.

With `WSimply`, we have been able to find the optimal solution with the three solving approaches[9] (using a 2.6 GHz Intel® Core™ i5) taking 12.27 s with the yices solving approach, 31.21 s with the core solving approach, and 126.00 s with the dico solving approach.

7.1.2 NSPLib instances

In order to evaluate the effects of the homogeneity meta-constraints on the quality of the solutions and the solving times, we have conducted an empirical study of some instances from the NSPLib. The NSPLib is a repository of thousands of NRP instances, grouped in different sets and generated using different complexity indicators: size of the problem (number of nurses, days or shift types), shifts coverage (distributions of the number of nurses needed) and nurse preferences (distributions of the preferences over the shifts and days). Details can be found in [35].

In order to reduce the number of instances to work with, we have focused on the N25 set, which contains 7920 instances. Since the addition of homogeneity meta-constraints in these particular instances significantly increases their solving time, we have ruled out the instances taking more than 60 s to be solved with `WSimply` without the meta-constraints. The final chosen set consists of 5113 instances. The N25 set has the following settings:

- Number of nurses: 25
- Number of days: 7
- Number of shift types: 4 (including the free shift)
- Shift covers: minimum number of nurses required for each shift and day.
- Nurse preferences: a value between 1 and 4 (from most desirable to least desirable) for each shift and day, for each nurse.

The NSPLib also has several complementary files with more precise information like minimum and maximum number of days that a nurse should work, minimum and maximum number of consecutive days, etc. We have considered the most basic case (case 1) which only constrains that

- the number of working days of each nurse must be exactly 5.

With the previous information we propose the NRP modelling of Fig. 4, where we have as hard constraints the shift covers and the number of nurse working days, and as soft constraints the nurse preferences.

In the following we report the results of several experiments performed with `WSimply` over the set of 5113 chosen instances of the NRP, using a cluster with nodes with CPU speed 1 GHz and 500 MB of RAM, and with a timeout of 600 s. We tested the three solving approaches (dico, yices and core). The times appearing in the tables are for the core approach, which was the one giving best results for this problem.

Table 1 shows the results of the chosen 5113 instances from the N25 set without homogeneity meta-constraints.

We can observe that if we only focus on minimising the cost of the violated constraints, we can penalise some nurses much more than others. For instance, we

[9]See Section 6.5.

```
Problem:nrp
 Data
   int n_nurses;
   int n_days;
   int n_shift_types;
   int covers[n_days, n_shift_types];
   int prefs[n_nurses, n_days, n_shift_types];
   int min_shifts;
   int max_shifts;
 Domains
   Dom dshifts_types = [1..n_shift_types];
   Dom dshifts  = [min_shifts..max_shifts];
   Dom dnurses = [0..n_nurses];
 Variables
   IntVar nurse_day_shift[n_nurses, n_days]::dshifts_types;
   IntVar nurse_working_shifts[n_nurses]::dshifts;
   IntVar day_shift_nurses[n_days, n_shift_types]::dnurses;
 Constraints
   %%% Every nurse must work only one shift per day.
   %%% This constraint is implicit in this modelling.

   %%% The minimum number of nurses per shift and day must be covered.
   %%% Variables day_shift_nurses[d,st] will contain the number of nurses working
   %%%  for every shift and day.
   Forall(d in [1..n_days], st in [1..n_shift_types]) {
     Count([nurse_day_shift[n,d] | n in [1..n_nurses]], st, day_shift_nurses[d,st]);
   };
   [day_shift_nurses[d,st] >= covers[d,st] | d in [1..n_days], st in [1..n_shift_types]];
   %%% Nurse preferences are desirable but non-mandatory.
   %%% Each preference is posted as a soft constraint with
   %%%  its label (#prefs[n,d,st]) and a violation cost according to prefs[n,d,st].
   Forall(n in [1..n_nurses], d in [1..n_days], st in [1..n_shift_types]) {
     #prefs[n,d,st]: (Not (nurse_day_shift[n,d] = st)) @ {prefs[n,d,st]};
   };
   %%% The minimum and maximum number of working days of each nurse
   %%%  must be between bounds (i.e. the domain of nurse_working_shifts[n]).
   Forall(n in [1..n_nurses]) {
     Count([ nurse_day_shift[n,d] <> n_shift_types | d in [1..n_days] ],
           True, nurse_working_shifts[n]);
   };
```

Fig. 4 `WSimply` model for the NRP

could assign the least preferred shifts to one nurse while assigning the most preferred shifts to others. From the results in Table 1, we observe that the mean of absolute differences is 7.71 while the mean cost per nurse is around 9.67, which shows that the assignments are not really fair. In order to enforce fairness, we can extend the model of Fig. 4 by adding homogeneity meta-constraints over the soft constraints on nurse preferences, as shown in Fig. 5.

Table 1 Results of 5113 instances from the N25 set, with soft constraints on nurse preferences (without meta-constaints)

	μ	σ	Time	Cost	Abs. diff.	Rel. diff.
Normal	9.67	1.83	6.46	241.63	7.71	9.42

μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; *Time*: mean solving time (in seconds); *Cost*: mean optimal cost; *Abs. diff.*: mean of differences between maximal and minimal costs; *Rel. diff.*: mean of differences between relative percentual maximal and minimal costs

```
%%% Ask for homogeneity with factor F, over the lists of
%%%  soft constraints on nurse preferences.
homogeneousAbsoluteWeight([ [ prefs[n,d,st] | d in [1..n_days],
                              st in [1..n_shift_types] ]
                            | n in [1..n_nurses] ], F);
```

Fig. 5 `WSimply` constraints to add to the NRP model in order to ask for homogeneity with factor `F` in the solutions

Table 2 shows the results after adding to the model of Fig. 4 the meta-constraint of Fig. 5, with factor `F` = 5, while Table 3 shows the results for the same meta-constraint with factor `F` = 10. Note that this factor represents the maximal allowed difference between the penalisation of the most penalized nurse and the least penalised nurse. From Table 1 we know that the mean of these differences among the chosen NRP instances is 7.71. The first row (Absolute 5) shows the results for the solved instances (2478 out of 5113) within the timeout. The second row shows the results without the meta-constraint, for the solved instances (i.e., it is like Table 1 but restricted to these 2478 instances).

As we can observe, we reduce the absolute difference average from 4.81 to 4.32, which is a bit more than 10 %. In particular, we reduce the absolute difference between the most penalised nurse and the least penalised nurse in 892 instances out of 2478. In contrast, the average penalisation per nurse increases from 8.80 to 8.92, but this is just 1.36 %. The average global cost also increases, but only from 220.03 to 222.89. Hence, it seems reasonable to argue that it pays off to enforce homogeneity in this setting, at least for some instances. However, when homogeneity is enforced the solving time increases, since the instances become harder (there are 2635 instances which could not be solved within the timeout).

The conclusion is that an homogeneity factor `F` = 5 may be too restrictive. Therefore, we repeated the experiment but with a factor `F` = 10. The results are shown in Table 3.

In this case only 946 out of 5113 could not be solved within the timeout. Although fewer instances are improved (377) the difference in the solving time really decreases and the mean of the best lower bounds for the unsolved instances is closer to the optimal value of the original instances. This suggests that it is possible to find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

Depending on the preferences of the nurses, the absolute difference may not be a good measure for enforcing homogeneity. Nurse preferences are weighted with

Table 2 Results when adding the `homogeneousAbsoluteWeight` meta-constraint with factor 5

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 5	8.92	0.96	43.28	222.89	272.93	4.32	892
Normal	8.80	1.07	5.98	220.03	261.94	4.81	–

Statistics for the 2478 solved instances with a timeout of 600 s. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; *Time*: mean solving time (in seconds); *Cost*: mean optimal cost; *Cost (TO)*: mean of best lower bounds for those instances that exceeded the timeout; *Abs. diff.*: mean of differences between maximal and minimal costs; *#improved*: number of instances with improved absolute difference

Table 3 Results when adding the `homogeneousAbsoluteWeight` meta-constraint with factor 10

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 10	9.32	1.46	13.86	233.01	285.83	6.29	377
Normal	9.31	1.47	6.16	232.83	280.40	6.35	–

Statistics for the 4167 solved instances with a timeout of 600 s. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; *Time*: mean solving time (in seconds); *Cost*: mean optimal cost; *Cost (TO)*: mean of best lower bounds for those instances that exceeded the timeout; *Abs. diff.*: mean of differences between maximal and minimal costs; *#improved*: number of instances with improved absolute difference

a value between 1 and 4 (from most desirable to least desirable shifts). Imagine a nurse who tends to weight with lower values than another. Then, even if this nurse has many unsatisfied preferences, her total penalisation could be lower than that of one of the other nurses with fewer unsatisfied preferences. Therefore, it seems more reasonable to compute the relative difference, as it allows the relative degree of unsatisfied preferences to be compared.

Table 4 shows the results for the meta-constraint `homogeneous PercentWeight` with factor 6, which means that the relative percent difference between the most penalised nurse and the least penalised nurse must be less than or equal to 6. The first row (Percent 6) shows the results for the solved instances (2109 out of 5113) within the timeout. The second row shows the results without the meta-constraint for those solved instances.

The mean of the percent differences is reduced from 7.76 to 5.26, which is almost 32 %. In particular, we reduce the percent difference between the most penalised nurse and the least penalized nurse in 1875 instances out of 2109. The average penalisation per nurse increases from 9.14 to 9.39, just 2.74 %, and the average global cost only increases from 228.56 to 234.72. However, the average solving time increases from 5.27 to 89.47 s for the solved instances. In fact, the solving time increases, no doubt, by much more than this on average if considering the timed-out instances.

As with the experiments for the absolute difference, we have conducted more experiments, in this case increasing the factor from 6 to 11. The results are reported in Table 5. In this case, only 492 out of 5113 could not be solved within the timeout. The number of improved instances decreases but the solving time improves. Therefore, with the `homogeneousPercentWeight` meta-constraint we can also find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

Table 4 Results when adding the `homogeneousPercentWeight` meta-constraint with factor 6

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 6	9.39	1.10	89.47	234.72	263.06	5.26	1875
Normal	9.14	1.30	5.27	228.56	250.81	7.72	–

Statistics for the 2109 solved instances with a timeout of 600 s. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; *Time*: mean solving time (in seconds); *Cost*: mean optimal cost; *Cost (TO)*: mean of best lower bounds for those instances that exceeded the timeout; *Rel. diff.*: mean of differences between relative percentual maximal and minimal costs; *#improved*: number of instances with improved relative difference

Table 5 Results when adding the `homogeneousPercentWeight` meta-constraint with factor 11

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 11	9.62	1.67	33.28	240.38	269.14	8.42	1592
Normal	9.57	1.71	5.51	239.23	264.20	9.04	–

Statistics for the 4621 solved instances with a timeout of 600 s. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; *Time*: mean solving time (in seconds); *Cost*: mean optimal cost; *Cost (TO)*: mean of best lower bounds for those instances that exceeded the timeout; *Rel. diff.*: mean of differences between relative percentual maximal and minimal costs; *#improved*: number of instances with improved relative difference

7.2 Soft balanced academic curriculum problem

The *Balanced Academic Curriculum Problem* (BACP) consists of assigning courses to academic periods while satisfying prerequisite constraints between courses and balancing the workload (in terms of credits) and the number of courses in each period [10, 19]. In particular, given

– a set of courses, each of them with an associated number of credits representing the academic effort required to successfully follow it,
– a set of periods, with a minimum and maximum bound both on the number of courses and number of credits assigned to each period,
– and a set of prerequisites between courses stating that, if a course c has as prerequisite a course d, then d must be taught in a period previous to the one of c,

the goal of the BACP is to assign a period to every course which satisfies the constraints on the bounds of credits and courses per period, and the prerequisites between courses. In the optimisation version of the problem, the objective is to improve the balance of the workload (amount of credits) assigned to each period. This is achieved by minimising the maximum workload of the periods.

There may be situations where the prerequisites make the instance unsatisfiable. We propose to deal with unsatisfiable instances of the decision version of the BACP by relaxing the prerequisite constraints, i.e., by turning them into soft constraints. We allow the solution to violate a prerequisite constraint between two courses but then, in order to reduce the pedagogical impact of the violation, we introduce a new hard constraint, the corequisite constraint, enforcing both courses to be assigned to the same period. We call this new problem *Soft Balanced Academic Curriculum Problem* (SBACP).

The goal of the SBACP is to assign a period to every course which minimises the total amount of prerequisite constraint violations and satisfies the conditionally introduced corequisite constraints, and the constraints on the number of credits and courses per period.

In Fig. 6 we propose a modelling of the SBACP using `WSimply`. In order to obtain instances of the SBACP, we have over-constrained the BACP instances from the MiniZinc [28] repository, by reducing the number of periods to four, and proportionally adapting the bounds on the workload and number of courses in each period. With this reduction on the number of periods, we have been able to turn all these instances into unsatisfiable.

```
Problem:sbacp
 Data
    int n_courses;
    int n_periods;
    int load_per_period_lb;
    int load_per_period_ub;
    int courses_per_period_lb;
    int courses_per_period_ub;
    int course_load[n_courses];
    int n_prereqs;
    int prereqs[n_prereqs,2];
 Domains
    Dom dperiods=[1..n_periods];
    Dom dload=[load_per_period_lb..load_per_period_ub];
    Dom dcourses=[courses_per_period_lb..courses_per_period_ub];
 Variables
    IntVar course_period[n_courses]::dperiods;
    IntVar period_load[n_periods]::dload;
    IntVar period_courses[n_periods]::dcourses;
 Constraints
 %%% Hard Constraints
 %%%
 %%% Every course must be assigned to exactly one period.
 %%% This constraint is implicit in this modelling.
 %%%
 %%% The number of courses per period must be within bounds
 %%%  (i.e. the domain of period_courses[p]).
     Forall(p in [1..n_periods]) {
       Count([ course_period[c] | c in [1..n_courses] ], p, period_courses[p]);
     };
 %%% The workload in each period must be within bounds
 %%%  (i.e. the domain of period_load[p]).
     Forall(p in [1..n_periods]) {
       Sum([If_Then_Else(course_period[c] = p) (course_load[c]) (0)
            | c in [1..n_courses]], period_load[p]);
     };
 %%% If a prerequisite is violated then the corequisite constraint is mandatory.
     Forall(np in [1..n_prereqs]) {
       Not(pre[np])
       Implies (course_period[prereqs[np,1]] = course_period[prereqs[np,2]]);
     };
 %%% Soft Constraints
 %%%
 %%% Prerequisites are desirable but non-mandatory.
 %%% Each prerequisite (np) is posted as a soft constraint with
 %%%  its label (pre[np]) and a violation cost of 1.
     Forall(np in [1..n_prereqs]) {
       #pre[np]: (course_period[prereqs[np,1]] > course_period[prereqs[np,2]]) @ 1;
     };
```

Fig. 6 WSimply model for the SBACP

In the following we present several experiments with WSimply over the obtained SBACP instances, using a 2.6 GHz Intel® Core™ i5, with a timeout of 600 s.

The best solving approach for this problem in our system is yices, closely followed by dico. The core solving approach is not competitive for this problem. The performance of the WPM1 algorithm strongly depends on the quality of the unsatisfiable cores the SMT solver is able to return at every iteration. This quality has to do, among other details, with the size of the core, the smaller the better, and the overlapping of the cores, the lower the better. For the SBACP instances, the SMT solver tends to return cores which involve almost all the soft clauses, i.e., they are as big as possible and they completely overlap. This clearly degrades the performance of the WPM1 algorithm.

Columns two to five of Table 6 show the results obtained by `WSimply` on our 28 instances. The second column shows the required CPU time in seconds (with the yices solving approach); the third column indicates the total amount of prerequisite violations, and the fourth and fifth columns show the maximum and minimum number of prerequisite constraint violations per course. This maximum and minimum exhibit the lack of homogeneity of each instance. Column six shows the time obtained by CPLEX for solving the SBACP instances.

Table 6 Results of the experiments on the SBACP instances without and with homogeneity

N.	Time	Cost	V. per c.		CPLEX	Homogeneity factor 1				Homogeneity factor 2			
			Max	Min		Time	Cost	V. per c.		Time	Cost	V. per c.	
								Max	Min			Max	Min
1	0.63	19	2	0	0.45	0.26	21	1	0	0.75	19	2	0
2	4.27	16	2	0	0.39	0.27	42	2	1	1.61	16	2	0
3	0.78	17	2	0	0.83	0.26	19	1	0	1.70	17	2	0
4	32.93	28	4	0	0.69	0.26	Unsatisfiable			3.69	28	**2**	0
5	0.97	15	2	0	0.43	0.25	39	2	1	0.86	15	2	0
6	0.56	10	2	0	0.43	0.31	10	**1**	0	0.47	10	2	0
7	1.35	19	2	0	0.50	0.28	40	2	1	0.86	19	2	0
8	2.63	21	3	0	0.46	0.24	Unsatisfiable			0.56	23	2	0
9	5.44	27	3	0	0.92	0.22	Unsatisfiable			1.26	27	**2**	0
10	3.43	21	3	0	0.57	0.28	39	2	1	3.07	21	**2**	0
11	10.23	22	3	0	0.59	0.29	38	2	1	2.29	22	**2**	0
12	18.11	27	3	0	0.66	0.30	47	2	1	4.30	27	**2**	0
13	1.29	14	3	0	0.32	0.28	17	1	0	0.72	15	2	0
14	0.47	17	2	0	0.40	0.44	33	2	1	0.34	17	2	0
15	0.17	6	2	0	0.20	0.45	28	2	1	0.26	6	2	0
16	1.61	15	2	0	0.31	0.29	15	**1**	0	1.10	15	2	0
17	10.72	23	5	0	0.66	0.24	Unsatisfiable			0.24	Unsatisfiable		
18	2.93	20	3	0	0.54	0.23	Unsatisfiable			1.09	20	**2**	0
19	0.43	16	2	0	0.37	0.25	39	2	1	0.41	16	2	0
20	3.71	15	2	0	0.59	0.49	15	**1**	0	3.58	15	2	0
21	1.93	14	2	0	0.47	0.25	20	1	0	0.61	14	2	0
22	0.74	15	2	0	0.43	0.31	17	1	0	0.55	15	2	0
23	2.18	20	1	0	0.63	0.28	20	1	0	2.33	20	1	0
24	0.22	7	2	0	0.30	0.32	9	1	0	0.30	7	2	0
25	3.03	13	2	0	0.33	0.52	14	1	0	1.58	13	2	0
26	0.23	5	1	0	0.21	0.38	5	1	0	0.35	5	1	0
27	1.09	17	2	0	0.43	0.25	21	1	0	1.48	17	2	0
28	0.19	10	2	0	0.28	0.26	11	1	0	0.34	10	2	0
T.	1.48	451			0.44	0.28	209			0.86	3		

Numbers in boldface denote instance improvements while maintaining the same cost. The last row shows the median of CPU solving time and the sum of the costs found; in the homogeneity cases we show the aggregated increment of the cost with respect to the original instances

```
%%% We enforce homogeneity with factor F, over the lists of
%%%  soft constraints on prerequisites for each course.
homogeneousAbsoluteNumber([ [ pre[np] | np in [1..n_prereqs], prereqs[np,1] = c ]
                            | c in [1..n_courses] ], F);
```

Fig. 7 `WSimply` constraints to add to the model for the SBACP in order to ask for homogeneity with factor `F` in the solutions

As we can observe, there are instances which have courses with three, four, and even five prerequisite constraint violations, as well as courses with zero violations. It would be more egalitarian to obtain solutions where the difference in the number of prerequisite constraint violations between courses is smaller. Thanks

Table 7 Results when adding the `multiLevel` meta-constraint to deal with the optimisation version of the SBACP, minimising the maximum workload per period and the maximum number of courses per period

N.	Homogeneity factor 2					MLevel: prereq, workload					MLevel: prereq, wl, courses				
	Time	c./p.		wl/p.		Time	c./p.		wl/p.		Time	c./p.		wl/p.	
		Min	Max	Min	Max		Min	Max	Min	Max		Min	Max	Min	Max
1	0.75	11	14	51	87	23.60	12	14	65	**66**	32.68	12	**13**	65	66
2	1.61	12	13	68	77	11.06	11	15	70	**71**	9.33	12	**13**	70	71
3	1.70	12	14	62	72	23.59	11	15	67	**68**	17.00	11	**14**	66	68
4	3.69	11	17	52	112	16.09	11	15	74	**77**	5.29	12	**13**	72	77
5	0.86	11	15	41	81	6.20	9	16	59	**63**	8.67	9	**15**	58	63
6	0.47	11	15	49	86	2.35	12	13	59	**60**	2.93	12	13	59	60
7	0.86	6	17	32	104	9.73	11	14	65	**66**	59.32	10	14	65	66
8	0.56	9	16	43	86	1.92	10	16	58	**66**	2.66	12	**13**	61	66
9	1.26	8	16	39	109	15.28	12	13	74	**77**	5.00	12	13	74	77
10	3.07	8	15	33	79	26.00	11	16	58	**66**	12.19	12	**14**	57	66
11	2.29	10	15	46	88	12.26	11	14	68	**68**	45.67	12	**13**	68	68
12	4.30	6	16	37	100	97.92	10	17	69	**70**	195.52	10	17	69	70
13	0.72	6	18	44	98	10.16	10	15	71	**72**	16.06	11	**13**	71	72
14	0.34	9	18	40	106	1.18	10	16	65	**69**	1.79	10	16	65	69
15	0.26	5	16	46	92	13.05	11	16	72	**72**	46.65	11	**13**	72	72
16	1.10	9	17	50	79	61.86	11	14	61	**63**	108.07	12	**13**	61	63
17	0.24	Unsatisfiable				0.59	Unsatisfiable				0.67	Unsatisfiable			
18	1.09	10	15	62	92	15.25	12	13	74	**75**	19.96	12	13	74	75
19	0.41	8	19	51	99	8.65	11	14	67	**68**	15.94	11	**13**	67	68
20	3.58	10	16	54	112	53.86	10	14	70	**75**	38.80	10	14	70	75
21	0.61	9	18	42	94	2.27	11	14	65	**65**	2.51	12	**13**	65	65
22	0.55	10	16	57	105	2.18	11	14	75	**77**	1.76	12	**13**	75	77
23	2.33	20	25	25	128	103.47	11	15	67	**68**	173.60	11	**13**	67	68
24	0.30	12	14	57	81	0.71	12	13	63	**78**	2.65	12	13	64	78
25	1.58	9	16	44	87	31.86	12	14	70	**70**	26.15	12	**13**	70	70
26	0.35	5	18	24	102	9.28	11	14	51	**78**	8.01	10	14	61	78
27	1.48	10	15	57	96	8.19	11	15	81	**81**	23.60	11	**14**	81	81
28	0.34	9	15	42	101	2.30	11	15	69	**70**	4.28	11	**14**	68	70
T.	0.86		439		2553	10.61				−654	14.07		−27		

Numbers in boldface denote improvements. Timeout is 600s. The last row shows the median of the solving time and the improvements on the aggregated maximums of the workload and number of courses per period thanks to the `multiLevel` meta-constraint

```
    int load_distance = load_per_period_ub - load_per_period_lb;
%%% load_bound is the upper bound of the load of all periods.
    IntVar load_bound::dload;
    Forall(p in [1..n_periods]) {
      period_load[p] =< load_bound;
    };
%%% We post as soft constraints each unit of distance between
%%%  load_bound and load_per_period_lb.
    Forall(d in [1..load_distance]) {
      #bala[d]: (load_bound < (load_per_period_lb + d)) @ {_};
    };
%%% We compose the new objective function with these two components,
%%%  being more important the prerequisites than the minimization of the
%%%  maximum load per period.
    multiLevel([[pre[np] | np in [1..n_prereqs]],
                [bala[d] | d in [1..load_distance]]]);
%%% Undefined weight of prerequisites soft constraints must be the same
    samePriority([pre[np] | np in [1..n_prereqs]]);
%%% Undefined weight of load distance soft constraints must be the same
    samePriority([bala[d] | d in [1..load_distance]]);
```

Fig. 8 Extension to minimise the maximum workload (amount of credits) of periods

to the homogeneousAbsoluteNumber meta-constraint we can easily enforce this property of the solutions, as shown in Fig. 7.

Also in Table 6, we show the results obtained by WSimply of these instances with homogeneity factor $F = 1$ (second block of columns) and $F = 2$ (third block of columns). The homogeneity factor bounds the difference in the violation of prerequisite constraints between courses (1 and 2 in our experiments). For homogeneity with factor 1, there are 5 unsolvable instances and 9 instances that achieve homogeneity by increasing the minimum number of violations per course (from 0 to 1) with, in addition, a dramatic increase in the total number of violations (+209). Experiments with homogeneity factor 2 give different results in 9 instances, all of which, except one becoming unsatisfiable, and are effectively improved by reducing the maximum number of violations per course, and slightly increasing the total number of violations (+3). Interestingly, the solving time has been improved when adding homogeneity.

By way of guidance a comparison between WSimply and CPLEX has been done only over the basic SBACP instances since CPLEX does not have any meta-constraint. WSimply exhibits a reasonably good performance in taking 1.48 s in median against the 0.44 s of CPLEX.

The first block of columns of Table 7 shows the minimum and maximum number of courses per period and the minimum and maximum workload per period, for each considered instance, when asking for homogeneity with factor 2 on the number of prerequisite violations.[10] As we can see, the obtained curricula are not balanced enough with respect to the number of courses per period and the workload per period. Therefore, we propose considering the optimisation version of SBACP by extending the initial modelling and using the multiLevel meta-constraint in order to improve the balancing in the workload and the number of courses per period (let us recall that the homogeneity meta-constraint on the number of violations

[10]We have chosen homogeneity factor 2 to continue with the experiments since, with factor 1, the number of violations increases in almost all instances.

introduced so far is hard). In Fig. 8 we show how we have implemented this extension for the workload (for the number of courses it can be done analogously). We also must set the weights of the prerequisite soft constraints to undefined to let the `multiLevel` meta-constraint compute them.

The idea of this encoding is to minimise the maximum workload per period using soft constraints. Column eleven (wl/p. Max) shows the improvement in the maximum workload per period obtained when introducing its minimisation with the `multiLevel` meta-constraint. Column fourteen (c./p. Max) shows the improvement in the the maximum number of courses per period obtained when adding its minimisation as the next level in the `multiLevel` meta-constraint.

8 Conclusions

We have introduced a new framework, called `WSimply`, which fills the gap between CSP and SMT regarding over-constrained problems. A new modelling language has been introduced for the intensional description of over-constrained problems. This language reasonably reduces the work needed to model this kind of problem and, at the same time, makes the models easier to read. In addition, the inclusion of meta-constraints also increases the capability to easily model several real-world problems. We have implemented some of the best-known meta-constraints from the literature, and we have extended some of them proposing variants and alternative meta-constraints. In particular, we have applied these meta-constraints to two well-known problems, the NRP and a variant of the BACP, showing how to improve the quality of the solutions and even the solving time for some cases. `WSimply` is the first declarative modelling language with such a high level of expressiveness for WCSP and meta-constraints.

Although in the preliminary comparision between CPLEX and SMT solving the SBACP, the performance of CPLEX is a bit better than SMT, the usage of SMT solvers in our solving strategies is a promising choice, since several constraints, once described intensionally, can be potentially more efficiently handled. We plan to incorporate newer algorithms from the MaxSAT community and adapt them to solve weighted SMT formulas.

We also plan to extend our framework with support for integer programming techniques, which may be more suitable for some problems where the Boolean structure is much less important than the arithmetic expressions.

Finally, we want to comment that we have also proposed a similar extension to deal with WCSPs for MiniZinc in [3], to which we could easily provide a solving mechanism based on SMT and WSMT as is done for `WSimply`.

Acknowledgement We thank the anonymous referees for their helpful comments and suggestions.

References

1. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M. (2011). A proposal for solving Weighted CSPs with SMT. In *Proceedings of the 10th international workshop on constraint modelling and reformulation (ModRef 2011)* (pp. 5–19).

2. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M. (2011). Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Proceedings of the 9th symposium on abstraction, reformulation and approximation (SARA 2011)* (pp. 2–9).
3. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M. (2011). W-MiniZinc: A proposal for modeling weighted CSP with MiniZinc. In *Proceedings of the 1st international workshop on MiniZinc (MZN 2011)*.
4. Ansótegui, C., Bonet, M.L., Levy, J. (2009). Solving (weighted) partial MaxSAT through satisfiability testing. In *Proceedings of the twelfth international conference on theory and applications of satisfiability testing (SAT 2009). LNCS* (Vol. 5584, pp. 427–440). Springer.
5. Barrett, C., Stump, A., Tinelli, C. (2010). The SMT-LIB standard: Version 2.0. In A. Gupta, & D. Kroening (Eds.), *Proceedings of the 8th international workshop on satisfiability modulo theories*. UK: Edinburgh.
6. Biere, A. (2008). Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation, 4*(2–4), 75–97.
7. Bofill, M., Palahí, M., Suy, J., Villaret, M. (2009). SIMPLY: A compiler from a CSP modeling language to the SMT-LIB format. In *Proceedings of the eighth international workshop on constraint Modelling and Reformulation (ModRef 2009)* (pp. 30–44).
8. Bofill, M., Palahí, M., Suy, J., Villaret, M. (2012). Solving constraint satisfaction problems with SAT modulo theories. *Constraints, 17*(3), 273–303.
9. Burke, E.K., Causmaecker, P.D., Berghe, G.V., Landeghem, H.V. (2004). The state of the art of nurse rostering. *Journal of Scheduling, 7*(6), 441–499.
10. Castro, C., & Manzano, S. (2001). Variable and value ordering when solving balanced academic curriculum problems. In *6th annual workshop of the ERCIM working group on constraints*.
11. Cooper, M.C., DeGivry, S., Schiex, T. (2007). Optimal soft arc consistency. In *Proceedings of the 20th international joint conference on artificial intelligence (IJCAI 2007)* (pp. 68–73).
12. deGivry, S., Zytnicki, M., Heras, F., Larrosa, J. (2005). Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proceedings of the 19th international joint conference on artificial intelligence (IJCAI 2005)* (pp. 84–89).
13. Dutertre, B., & deMoura, L. (2006). A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th international conference on computer aided verification, CAV'06. LNCS* (Vol. 4144, pp. 81–94). Springer.
14. Dutertre, B., & deMoura, L. (2006). The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf.
15. Eén, N., & Sörensson, N. (2003). An extensible sat-solver. In E. Giunchiglia, & A. Tacchella (Eds.), *SAT. Lecture notes in computer science* (Vol. 2919, pp. 502–518). Springer.
16. Freuder, E.C., & Wallace, R.J. (1992). Partial constraint satisfaction. *Artificial Intelligence, 58*(1–3), 21–70.
17. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I. (2008). Essence: a constraint language for specifying combinatorial problems. *Constraints, 13*(3), 268–306.
18. Gent, I., Miguel, I., Rendl, A. (2010). Optimising quantified expressions in constraint models. In *Proceedings of the 9th international workshop on constraint modelling and reformulation (ModRef 2010)*.
19. Hnich, B., Kiziltan, Z., Walsh, T. (2002). Modelling a balanced academic curriculum problem. In *Proceedings of the fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems (CPAIOR 2002)* (pp. 121–131).
20. Larrosa, J., & Meseguer, P. (2002). Partition-based lower bound for max-CSP. *Constraints, 7*, 407–419.
21. Larrosa, J., Meseguer, P., Schiex, T. (1999). Maintaining reversible DAC for max-CSP. *Artificial Intelligence, 107*(1), 149–163.
22. Larrosa, J., & Schiex, T. (2004). Solving weighted CSP by maintaining arc-consistency. *Artificial Intelligence, 159*(1–2), 1–26.
23. Mahajan, Y.S., Fu, Z., Malik, S. (2004). Zchaff2004: An efficient SAT solver. In H.H. Hoos, & D.G. Mitchell (Eds.), *Proceedings of the 7th international conference on theory and applications of satisfiability testing (SAT 2004). LNCS* (Vol. 3542, pp. 360–375). Springer.
24. Manquinho, V.M., Silva, J.P.M., Planes, J. (2009). Algorithms for weighted Boolean optimization. In *Proceedings of the twelfth international conference on theory and applications of satisfiability testing (SAT 2009). LNCS* (Vol. 5584, pp. 495–508). Springer.
25. Meseguer, P., Rossi, F., Schiex, T. (2006). Soft constraints. In F. Rossi, P. van Beek, T. Walsh (Eds.), *Handbook of constraint programming* (chapter 9). Elsevier.

26. Métivier, J.-P., Boizumault, P., Loudni, S. (2009). Solving nurse rostering problems using soft global constraints. In *Proceedings of the 15th international conference on principles and practice of constraint programming (CP 2009)*. *LNCS* (Vol. 5732, pp. 73–87). Springer.
27. Michel, L., See, A., VanHentenryck, P. (2009). Parallel and distributed local search in COMET. *Computers and Operations Research, 36*, 2357–2375.
28. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th international conference on principles and practice of constraint programming (CP 2007)*. *LNCS* (Vol. 4741, pp. 529–543). Springer.
29. Nieuwenhuis, R., & Oliveras, A. (2006). On SAT modulo theories and optimization problems. In *Proceedings of the 9th international conference on theory and applications of satisfiability testing (SAT 2006)*. *LNCS* (Vol. 4121, pp. 156–169). Springer.
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C. (2006). Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM, 53*(6), 937–977.
31. Palahí, M. (2010). Eines basades en la lògica per a modelatges i resolució de problemes combinatoris. Master's thesis, Universitat de Girona/Universitat Politècnica de Catalunya, Spain.
32. Petit, T., Regin, J.C., Bessiere, C. (2000). Meta-constraints on violations for over constrained problems. In *12th IEEE international conference on tools with artificial intelligence (ICTAI 2000)* (pp. 358–365).
33. Roussel, O., & Lecoutre, C. (2009). XML representation of constraint networks: format XCSP 2.1. *Computing Research Repository, abs/0902.2362*. http://arxiv.org/abs/0902.2362.
34. Sebastiani, R. (2007). Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation, 3*(3–4), 141–224.
35. Vanhoucke, M., & Maenhout, B. (2007). NSPLib—a nurse scheduling problem library: A tool to evaluate (meta-) heuristic procedures. In *Operational research for health policy making better decisions* (pp. 151–165).