A Proposal for Solving Weighted CSPs with SMT*

Carlos Ansótegui[1], Miquel Bofill[2], Miquel Palahí[2], Josep Suy[2], and Mateu Villaret[2]

[1] Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida, Spain
carlos@diei.udl.cat
[2] Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona, Spain
{mbofill,mpalahi,suy,villaret}@ima.udg.edu

Abstract. We introduce a weighted CSP modeling language that allows to represent over-constrained problems by means of hard and soft constraints. In contrast to other well-known approaches such as XCSP, which are mainly designed for extensional representation of goods or no-goods and hence do not provide much declarative facilities, our approach aims to follow the declarative syntax style of MiniZinc. Therefore, we fill the gap of a CSP modeling language allowing the intensional representation of weighted CSPs. In addition, our language has built-in support for some meta-constraints, such as priority and homogeneity.
We propose the compilation of the obtained descriptions either into SMT and solving them by means of binary search optimization, or its compilation into MaxSMT and using core based algorithms borrowed from weighted MaxSAT.

1 Introduction

A Constraint Satisfaction Problem (CSP) is a problem where the goal is to determine whether there exists an assignment of values to a set of variables which satisfies a given set of constraints. CSPs are either decision or optimization problems. In the case of optimization, one typically seeks for a solution which minimizes or maximizes the value of a given objective function. However, most of the developed frameworks for modeling and solving CSPs are not able to deal with over-constrained problems, where a solution satisfying all the constraints may not exist and where, roughly, one must seek for a solution which minimizes the cost associated to the unsatisfied constraints or, dually, maximizes certain preferences. These problems arise in many real applications. For this reason, in the last few years the CSP framework has been augmented with so-called soft

* Partially supported by the Spanish Ministry of Science and Innovation through the projects SuRoS (ref. TIN2008-04547), TIN2010-20967-C04-01/03 and TIN2009-14704-C03-01.

constraints, with which it is possible to express preferences among solutions in problems allowing some degree of violation of constraints.

Several soft constraints frameworks have been proposed. For example, in Weighted Constraint Satisfaction Problems (WCSPs), one can distinguish between constraints which cannot be violated (hard constraints) and constraints which have a violation cost (soft constraints). Then the goal is to find a full assignment which satisfies all hard constraints and minimizes the aggregated cost of the violated soft constraints [12].

XCSP 2.1 [17] is an XML format which has been recently adopted in the CSP, MaxCSP and WCSP solver competitions. Although it allows to define constraints either in extension or intension, most of the available WCSP benchmarks written in XCSP are described in extension. Some tools supporting higher-level, less verbose languages such as MiniZinc [15], can output to XCSP. There also exist tools like TAILOR [9], which translate from XCSP to other declarative, solver-independent modeling languages such as ESSENCE' (a subset of ESSENCE [8]). However, to our knowledge, none of those higher-level languages includes direct support for WCSP.

In order to contribute to fill this gap, in this work we focus on the (intensional) specification and resolution of WCSP from a high-level language. In particular, we extend the medium-level constraint modeling language of `Simply` [3] to deal with costs associated to the violation of constraints. Our language is somehow inspired by the declarative high-level syntax style of MiniZinc. `Simply` solves decision and optimization CSP instances by compiling them into SMT and calling an external SMT solver. The extension we propose can deal with WCSP instances by either turning them into optimization CSP instances or, alternatively, by compiling them into weighted MaxSMT instances which are solved using core based algorithms borrowed from weighted MaxSAT. Furthermore, our language has built-in support for meta-constraints, covering all kinds described in [16]. Meta-constraints can be very helpful in the modeling process, since they allow us to abstract to a higher level. For example, one can impose certain priority among a set of soft constraints, instead of having to define a concrete weight for each of them.

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

2 Weighted CSP

When constraint programming faces real-word applications it is not strange to find that these are over-constrained and do not have any solution. In such situations, it is necessary to relax the problem in order to obtain some solution.

The most well-known approach is to find an assignment which minimizes the number of violated constraints (Maximal Constraint Satisfaction Problem, MaxCSP). There are many works about MaxCSP [7,11,10] but for many problems this approach is not the best solution. Sometimes it is better to violate certain constraints than others. For example, in the nurse rostering problem 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. Thus, the constraints have different priorities, and in some cases the degree of violation of the constraints can be important. This approach can be achieved by extending the classical CSP framework by associating weights (costs) to constraints. In the resulting Weighted Constraint Satisfaction Problems (WCSPs) [4,5], the goal is to find an assignment with minimum aggregated cost of the violated constraints.

Next we formally define constraint satisfaction and weighted constraint satisfaction problems.

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 that 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 (nogood 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* $v(x_i) \in d(x_i)$*.*

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

A (partial) assignment v satisfies a constraint $\langle\{x_{i_1}, \ldots, x_{i_k}\}, R_i\rangle$ *in C iff* $\langle v(x_{i_1}), \ldots, v(x_{i_k})\rangle \in R_i$*.*

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 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* $f_i : d(x_{i_1}) \times \cdots \times d(x_{i_k}) \to \mathbb{N}$ *is a* cost (weight) function *that maps tuples to its associated weight. The cost (weight) of a constraint* C_i *induced by an assignment v in which the variables of* $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ *take values* $b_{i_1}, \ldots, b_{i_k}$ *is* $f_i(b_{i_1}, \ldots, b_{i_k})$*.*

An optimal solution to a WCSP instance is a complete assignment in which the sum of the costs of the constraints is minimal.

Definition 3. *The Weighted Constraint Satisfaction Problem (WCSP) for a WCSP instance consists in finding an optimal solution for that instance.*

3 Weighted SMT

In the last decades SAT solvers have spectacularly progressed 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. Thanks to those advances, nowadays best SAT solvers can tackle problems with hundreds of thousands of variables and millions of clauses.

An SMT instance is a generalization 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.

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 specialized solvers for a set of literals belonging to each theory. It is worth to note 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 Weighted SMT as follows:

A *weighted SMT clause* is a pair (C, w), where C is an SMT clause[3] and w is a natural number or infinity (indicating the penalty for falsifying C). A clause is called *hard* if the corresponding weight is infinity, otherwise the clause is called *soft*.

A *weighted (partial) MaxSMT 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. The Weighted MaxSMT problem for a weighted MaxSMT formula is the problem of finding an optimal assignment.

The Weighted MaxSMT problem for a weighted MaxSMT formula can be reformulated as an optimization problem, i.e., the maximization of a linear function subject to a set of constraints, as follows:

$$\text{maximize } b_1 \cdot w_1 + \cdots + b_m \cdot w_m \tag{1}$$

[3] In fact these are general SMT formulas, not necessarily being disjuctions of atoms.

subject to:

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

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

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

$$\text{minimize } \sum_{i=1}^{m} w_i - (b_1 \cdot w_1 + \cdots + b_m \cdot w_m) \tag{4}$$

We can extend our notion of Weighted MaxSMT formula by allowing the w_i to be linear arithmetic expressions. Note that the equivalent optimization problem now can have nonlinear objective functions. We will refer to these formulas as Generalized Weighted MaxSMT formulas.

4 Soft Constraints in `Max-Simply`

`Simply` was developed as a declarative programming system for easy modeling and solving of CSPs. `Simply` deals with CSP instances by translating them into SMT instances that are fed to an SMT solver. If the SMT solver finds a solution, this is translated back as a solution of the original CSP instance.

`Simply` is similar to other high-level languages such as ESSENCE and MiniZinc. We can specify the model and the data in separate files. In `Simply`, the model consists of six sections: data definition, domain definition, variable declaration, user defined predicate definition, constraint definition and optimization. Decision variables are Boolean or finite domain integer variables on which we can post all sorts of relational and logical constraints. The language has the useful declarative facility of comprehension lists that allows to produce concise and elegant modelings. Further details of the translation process to SMT and the architecture of the system can be found in [3][4].

4.1 The language

Many existing WCSP solving systems and languages consider an extensional approach, i.e., deal with instances consisting of an enumeration of good/no-good tuples for hard constraints and nogood tuples with a cost for soft constraints. Our proposal follows the other direction and aims to allow users to model intensionally the violation cost of their soft constraints.

[4] `Max-Simply` is available at `http://ima.udg.edu/Recerca/ESLIP/simply/`.

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 nogood tuples: $(x = 1, y = 1, 1), (x = 2, y = 1, 1)$ and $(x = 2, y = 2, 1)$. In `Max-Simply` we would express it as: $(x < y)$ @ $\{1\}$.

Simple weighted constraints The most basic type of weighted constraint in `Max-Simply` is of the form:

$$(\texttt{constraint}) \,@\, \{\texttt{expression}\};$$

where the value of `expression` is the weight (cost) of falsifying the associated `constraint`. The expression must be a linear integer arithmetic expression. It can either be evaluable at compile time, or contain decision variables. For example,

$$(\texttt{a} < \texttt{b}) \,@\, \{\texttt{10}\}; \qquad (\texttt{a} < \texttt{b}) \,@\, \{\texttt{a} - \texttt{b} + \texttt{1}\};$$

Note that by allowing to use decision variables in the cost expression we can encode the degree of violation of a constraint. Weights will always be positive integers.

Labeled weighted constraints

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

In this case, the `label` allows us to refer to the constraint inside other constraints. For example,

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

Meta-constraints In order to provide a higher level of abstraction in the modeling of over-constrained problems, in [16] several meta-constraints[5] are proposed. All of them are covered by our language. To this end, a third kind of weighted constraint is necessary:

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

where the underscore "_" denotes an undefined weight. The value of this undefined weight is computed at compile time according to the meta-constraints that refer to the label. This simplifies the modeling of the problem, since the user does not need to compute the concrete weights.

The meta-constraints can be related to the following:

[5] A meta-constraint is, roughly, a constraint on constraints.

1. **Priority.** A constraint has higher priority than another. For instance, if we have an activity to perform and worker 1 *doesn't want* to perform it while worker 2 *should not* perform it, then it is better to violate the first constraint than the second. In our language the following meta-constraints refer to this aspect:
   - `samePriority(`*List*`)`, where *List* is a list of labels of weighted constraints. With this meta-constraint we are stating that the constraints referred to in the *List* are soft constraints with the same priority.
   - `priority(`*List*`)`, where *List* is a list of labels of weighted constraints. With this meta-constraint we are stating that the constraint corresponding to the i-th label in *List* has higher priority than the constraint corresponding to the $(i+1)$-th label. In other words, it must be more costly to violate the i-th constraint than the $(i+1)$-th.
   - `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, we are stating that $weight_1 \geq weight_2 * n$.
2. **Degree of violation.** The cost of violating a constraint can be relative to some degree of violation. For instance, for a worker working more than five turns in a week, the violation cost could be increased by, e.g., one unit for each extra worked turn. Since decision variables can be used in the expressions defining the weights, these constraints can be easily stated with our language:

   $$(\texttt{worked\_turns} < 6) \ @ \ \{\texttt{base\_cost} + \texttt{worked\_turns} - 6\};$$

3. **Homogeneity.** Sometimes, the degree of violarion of a certain group of constraints is desired to be homogeneous. For instance, the number of days off for workers may be desirable to be as homogeneous as possible. We have the following meta-constraints related to homogeneity:
   - `atLeast(`*List*`,`p`)`, where *List* is a list of labels of weighted constraints and p is a real number in 0..1. This meta-constraint ensures that the ratio of constraints corresponding to the labels in *List* that are satisfied is at least p.
   - `homogeneous(`*ListOfLists*`,`p`)`, where *ListOfLists* is a list of lists of labels and p is a real number in 0..1. This meta-constraint ensures that, for each pair of lists in *ListOfLists*, the ratio of satisfied constraints in each list differs at most in p. For example,

     ```
     metaConstraint homogeneous([[A,B,C],[D,E,F],[G,H,I]],0.33);
     ```

     ensures that if constraints `A`, `B` and `C` are satisfied, then at least two constraints of each list `[D,E,F]` and `[G,H,I]` are satisfied.
4. **Dependence.** Particular configurations of violations entail the necessity to satisfy other constraints, that is, if a soft constraint is violated then another soft constraint must not, or a new constraint must be satisfied. For instance,

working the first or the last turn of the day is penalized, but if you work in the last turn of one day, then the next day you cannot work in the first turn. This could be succinctly stated as follows:

```
#A:(not_last_turn_day_1)@{w1}; #B:(not_first_turn_day_2)@{w2};
(Not A) Implies B;
```

As another example, if there is a penalization on working in the last turn, and for a worker this is violated on Monday and Tuesday, we may need to compensate these two successive violations by imposing that, on Wednesday, the worker must not work in the last turn. This can again be easily encoded by using implication:

```
#A:(not_last_turn_mon)@{w1}; #B:(not_last_turn_tue)@{w2};
(Not A And Not B) Implies NewConstraint;
```

where `NewConstraint` can be something such as `turn[wed] < last`.

4.2 Translation of meta-constraints

Our approach for managing the meta-constraints consists on removing all of them by their reformulation into `Max-Simply` constraints.

In order to deal with the *priority* meta-constraints, we create a system of linear inequations on the undefined weights of the soft constraints referenced by the meta-constraint. 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:

$$w_A = 3, w_B > 0, w_C > 0, w_D > 0,$$
$$w_A > w_B, w_B > w_C,$$
$$w_D \geq 2 \cdot w_B$$

This set of inequations is fed into an SMT solver[6] 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 solver would return us a model such as, e.g.,:

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

This allows to reformulate the original problem into an equivalent WCSP without undefined weights (which is solved as explained in Section 6):

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

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

Hence, with the meta-language, and thanks to this simple use of a solver at compile time, we free the user of the tedious task of thinking about concrete weights for encoding priorities. If the constraints turn out to be contradictory, then the solver will report that the resulting set of inequations is unsatisfiable, and the user will be warned about this fact at compile time.

We remark that, since the undefined weights need to be determined at compile time, no decision variables can be used in any weight expression involved with priority meta-constraints.

In order to deal with the *violation degree* meta-constraints, we create an auxiliary integer variable for each weight expression. These variables are restricted to be equal to zero if the constraint is satisfied, and equal to the weight expression if the constraint is falsified (see Equation 5 of Section 6).

We deal with the *homogeneity* meta-constraints by reifying the constraints on which we are applying the meta-constraint and constraining the number of satisfied constraints. For instance, the meta-constraint `atLeast(`*List*`,`*p*`)` is reformulated into:

```
Count(ListReif,True,n);
n >= p*len;
```

where `ListReif` is the list of Boolean variables resulting from reifying the list of constraints referenced in *List*, and `len` is the length of *List*. `Count(`*l*`,`*e*`,`*n*`)` is a global constraint that is satisfied if and only if there are exactly n occurrences of element e in list l.

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

```
Count(ListOfLists[1],True,n[1]);
Count(ListOfLists[2],True,n[2]);
...
Abs((n[1]*100 Div len[1])-(n[2]*100 Div len[2])) =< p*100;
Abs((n[1]*100 Div len[1])-(n[3]*100 Div len[3])) =< p*100;
...
Abs((n[l-1]*100 Div len[l-1])-(n[l]*100 Div len[l])) =< p*100;
```

where `l` is the length of *ListOfLists*, `Abs(`*exp*`)` computes the absolute value of *exp*, and `len`$[i]$ is the length of the i-th list of labels in *ListOfLists*.

Finally, we deal with the *dependence* meta-constraints by simply using the logical operators between constrains directly supported by `Max-Simply`.

5 Nurse Rostering Problem Example

One of the paradigmatic examples of over-constrained CSP is 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, that can be hard or soft, are usually defined by regulations, working practices and nurses preferences [14].

5.1 The GPost instance

We illustrate the use of meta-constraints on a simplified variant of the GPost NRP instance[7]. In this example we consider 4 weeks and 8 employees (4 full-timers and 4 part-timers), and two shift types (day and night). So we define a variable array `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 has a day shift, and `sh[i,d]=2` means that the nurse has a night shift. Shifts numbered from 1 to 4 refer to full-timers, and from 5 to 8 refer to part-timers.

Hard constraints. Full-timers work exactly 18 shifts in 4 weeks while part-timers only 10. In `Max-Simply` this can be encoded as follows:

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

Each employee works at most 4 night shifts, of which at most 3 are consecutive. In order to count the number of night shifts per worker we have to create another array of variables `tns[8]` with domain `[0..4]`.

```
Forall(i in [1..8]) {
  Count( [sh[i,d] | d in [1..28]], 2, tns[i]); }
//restrict the consecutive days
Forall(i in [1..8]) {
  Forall(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) } }
```

Note that the maximum number of night shifts is bounded by the domain.

Soft constraints. There is a penalization for a single night shift (first and last days of the roster are ignored for the sake of simplicity):

```
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 labels can be indexed as arrays. Also, we left the weight undefined, since we want to state that all of these constraints have the same priority. For doing this we only have to post the following meta-constraint:

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

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

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

```
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 than violating the `NSP` constraints, we use the following meta-constraint[8]:

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

A full-timer has to work 4 or 5 days per week. We want to consider the deviation as the violation degree:

```
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 the homogeneity meta-constraints to guarantee a minimum satisfaction on the free days that each nurse prefers. We first state, as a soft constraint of weight 1, each free day requested by each nurse being free[9]:

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

And, second, we can apply the following meta-constraints in order to guarantee a minimum of preferences satisfied and to obtain certain homogeneity in satisfaction among nurses, e.g.:

```
atLeast([PFD[i,f] | i in [1..8], f in [1..5]], 0.40);
homogeneous([[PFD[1,f]|f in [1..5]],
             [PFD[2,f]|f in [1..5]], ...], 0.50);
priority(AFD[1,1], PFD[1,1], 10);
```

6 Solving Approaches

The problem modeled in `Max-Simply` is translated into a Generalized Weighted MaxSMT instance, which represents an optimization problem, as described in Section 3. Once `Max-Simply` has converted the problem instance into a Generalized Weighted MaxSMT instance, this can be reformulated either into a sequence of SMT instances (successively restricting the value of the objective function) or

[8] Note that, by making the first constraints of the two lists have the same priority, we are making all the constraints of both lists have the same priority.

[9] We assume that each nurse has asked for 5 preferred free days.

into a single Weighted SMT instance. That would depend of the solving strategy to be used. Note that not all the SMT solvers provide support for solving Weighted SMT instances.

Regarding solving optimization problems exactly, there are two main approaches: branch and bound algorithms and decision-based algorithms. Branch and bound algorithms typically perform a tree search, refining lower and upper bounds that allow to reduce the search space. On the other hand, decision-based algorithms perform a sequence of decisional queries that bound the objective function. Informally, these queries state whether it is possible to evaluate the objective function to a value less or equal than a certain bound, subject to the original hard constraints of the problem.

`Max-Simply` is designed to use SMT solvers as a black box. Thus, decision-based algorithms become a natural candidate, since we can solve a Generalized Weighted SMT problem through a sequence of calls to an SMT solver. We actually know from the experience in the MaxSAT field that the solvers implementing these algorithms show a good performance on industrial instances.

Solving approach through translation to SMT In the following we describe the basic scheme of decision-based algorithms. A Generalized Weighted MaxSMT 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 cost smaller or equal to k (k plays the role of the bound that 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 $k \geq k_{opt}$, are satisfiable, while for $k < k_{opt}$ 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). 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 some 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 boost the efficiency of these approaches is to know whether we can exploit any additional information from the execution of the SMT solver for the next runs.

Now, we define how to encode φ_k as an SMT formula. One way to encode φ_k is to create a fresh integer variable o_i for each soft clause. This variable will either evaluate to weight w_i if the clause C_i is violated, or to 0 otherwise. Therefore, for each soft clause `Max-Simply` will generate the following constraints expressed as SMT clauses (instead of equations 2):

$$o_i \geq 0 \qquad c_i \vee (o_i > 0) \qquad (o_i = w_i) \vee (o_i = 0) \tag{5}$$

Note that the first clause forbids negative weights. The second and third clauses state that if c_i is falsified, then the cost is w_i. Since we are minimizing costs it will never happen that c_i is satisfied and the cost is greater than 0.

`Max-Simply` also introduces another integer variable, O, which represents the sum of the o_i variables, and will be bounded by k as follows:

$$O = \sum_{i=1}^{m} o_i \qquad\qquad O \leq k \tag{6}$$

The minimization of O plays the role of the minimization in equation 4. The set of hard constraints are added without modification to the SMT formula. Actually, once we have properly introduced the constraints on the o_i and O variables, we just need to add a new $O \leq k$ constraint at every call to the SMT solver. In this work, the strategy we use to determine which $O \leq k$ we add is a binary search. We will refer to this solving method as dico.

Solving approach through translation to Weighted SMT An alternative solving method is to convert a Generalized Weighted MaxSMT instance into a Weighted MaxSMT instance and to use any SMT solver that is able to handle these problems. Therefore, in order to be able to use these solvers or to use algorithms from the Weighted MaxSAT field, we need to translate our problem into one where the weights can only be constants. In essence, we just construct a Weighted MaxSMT instance almost as we constructed the SMT instances in the previous solving method, i.e., as clauses of equation 5. The difference is that instead of adding the hard constraints of equation 6, we add a set of soft clauses over each possible value of each o_i:

$$\bigwedge_{i=1..m} \Big(\bigwedge_{v_j \in V(o_i)} (o_i \neq v_j, v_j) \Big) \tag{7}$$

where $V(o_i)$ is the set of all > 0 possible values of o_i.

Instead of removing equations 6 and adding the clauses of equation 7, we could just remove $O \leq k$ and add the set of soft constraints $\bigwedge_{i=1}^{i=W} (O < i, 1)$, where W is the greatest value the objective function can be evaluated to (we use this method for our experiments). A more concise alternative could result from using the binary representation of W, adding the following soft constraints: $\bigwedge_{i=0}^{i<\lceil log_2(W+1) \rceil} (\neg b_i, 2^i)$, and the hard constraint: $\sum_{i=0}^{i<\lceil log_2(W+1) \rceil} 2^i \cdot b_i = O$.

Yices [6] offers a method to directly solve Weighted Max-SMT instances. We will refer to this solving method as yices. Since this is yet an immature research topic in SMT, we have extended the Yices framework by incorporating other algorithms from the MaxSAT field. In particular, we have implemented the algorithm WPM1 from [2,13], which is based on the detection of unsatisfiable cores. These are decision-based algorithms, where the parameter k ranges from 0 to k_{opt}. Then, for every UNSAT answer, they analyze the core of unsatisfiability of the formula returned by the SMT solver. This information is incorporated as redundant constraints into the next call to the SMT solver which help to boost the propagation. The performance of these algorithms heavily depends on the quality of the cores. Therefore, we have extended them with a heuristic that prioritizes those cores which involve constraints with higher weights. In our experiments, we refer to the method which uses the binary encoding of the objective function and this algorithm as core.

7 Benchmarks

We have modeled the GPost NRP instance with `Max-Simply` obtaining reasonably good performance w.r.t. the results on the same problem reported in [14]. They report 8 seconds finding an optimal solution of cost 3 with an ad hoc CPLEX search over a previously computed enumeration of all possible schedules for each nurse (2.83GHz Intel Core 2 Duo). They also report 234 seconds finding a non optimal solution of cost 8 with their generic local search method (VNS/LDS+CP) based on neighborhoods plus an exploration of the search space with CP and soft global constraints (2.8GHz Pentium IV). We have not found any optimal solution within 3600 seconds using the yices solving approach, but have found an optimal solution within 260 and 3291 seconds using the core and dico solving approaches respectively (2.6GHz Intel Core i7).

This is still a work in progress and we plan to model more instances within `Max-Simply` to obtain more evidences of the robustness of our proposal. Nevertheless, in order to extensively test the performance of our solving strategies, and since we have not found WCSP instances described intensionally, we decided to run our experiments on some sets of WCSP instances described extensionally[10]. In order to compare with a competitive WCSP solver we chose toulbar2 [11], which has been the winner at the WCSP category in several recent CSP competitions. Our solving approaches were: dico, yices and core, described in previous section. The toulbar system proved to be the best performing solver. However our SMT approaches performed reasonably well. Overall, the average percentage of solved instances for toulbar was 87% while for the whole SMT approach was 79%.

8 Conclusions

The framework we have presented, `Max-Simply`, fills the gap between CSP and SMT regarding over-constrained problems. The new modeling language we have introduced for the intensional description of over-constrained problems and the inclusion of meta-constraints increases the capability to easily model several real problems. The meta-constraints that we have implemented are the best-known ones from the literature. Also, 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. Nevertheless, we plan to make extensive benchmarking on several intensional over-constrained problems like nurse rostering instances.

We will also like to highlight that we have introduced a more general notion of Weighted MaxSMT formulas where the weights can be linear arithmetic expressions. In this sense, we have presented new encoding and solving techniques to address this new formalism.

[10] `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`
[11] `https://mulcyber.toulouse.inra.fr/projects/toulbar2/`

Finally, we want to remark that we have also proposed a similar extension to deal with weighted CSPs for MiniZinc in [1], to which we could easily provide a solving mechanism based on SMT and Max-SMT as is done for `Max-Simply`.

References

1. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. W-MiniZinc: A Proposal for Modeling Weighted CSP with MiniZinc. In *MZN 2011*, (to appear).
2. C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *SAT 2009*, volume 5584 of *LNCS*, pages 427–440. Springer, 2009.
3. M. Bofill, M. Palahí, J. Suy, and M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *ModRef 2009*, pages 30–44, 2009.
4. M. C. Cooper, S. De Givry, and T. Schiex. Optimal soft arc consistency. In *IJCAI 2007*, pages 68–73, 2007.
5. S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *IJCAI 2005*, pages 84–89, 2005.
6. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.
7. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21 – 70, 1992.
8. A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
9. I. Gent, I. Miguel, and A. Rendl. Optimising Quantified Expressions in Constraint Models. In *ModRef 2010*, 2010.
10. J. Larrosa and P. Meseguer. Partition-Based Lower Bound for Max-CSP. *Constraints*, 7:407–419, 2002.
11. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149 – 163, 1999.
12. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-Consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.
13. V. M. Manquinho, J. P. M. Silva, and J. Planes. Algorithms for Weighted Boolean Optimization. In *SAT 2009*, volume 5584 of *LNCS*, pages 495–508. Springer, 2009.
14. J.-P. Métivier, P. Boizumault, and S. Loudni. Solving Nurse Rostering Problems Using Soft Global Constraints. In *CP 2009*, volume 5732 of *LNCS*, pages 73–87. Springer, 2009.
15. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
16. T. Petit, J. C. Regin, and C. Bessiere. Meta-constraints on violations for over constrained problems. In *ICTAI 2000*, pages 358–365, 2000.
17. O. Roussel and C. Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *Computing Research Repository*, 2009.