W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc*

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 propose extending MiniZinc to W-MiniZinc, in order to deal with over-constrained problems by means of weighted CSP. In contrast to other well-known approaches such as XCSP, which are mainly designed for extensional representation of goods or nogoods and hence do not provide much declarative facilities, our proposal is intended for intensional representations. Therefore, we fill the gap of a high-level declarative modeling language allowing the intensional representation of weighted CSPs. In addition, our proposal incorporates support for some meta-constraints, such as priority and homogeneity.

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 the so-called soft constraints, with which it is possible to express preferences among solutions in problems allowing some degree of violation of constraints [16].

Several soft constraints frameworks have been proposed. For example, in Weighted CSP (WCSP), one can distinguish between constraints which cannot be violated (hard constraints) and constraints which have a violation cost

* 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.

(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 [11].

XCSP 2.1 [14] 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. The MiniZinc [12] distribution contains a tool that can output to XCSP. There also exist tools like TAILOR [8], which translate from XCSP to other declarative, solver-independent modeling languages such as ESSENCE' (a subset of ESSENCE [7]). However, to our knowledge, none of those higher-level languages directly supports WCSP[3].

In order to contribute to fill this gap, in this proposal we focus on the (intensional) specification and resolution of WCSPs from a high-level language. In particular, we extend the MiniZinc constraint modeling language to deal with costs associated to the violation of constraints, in an extension that we call W-MiniZinc. We show how the resulting instances can be translated into FlatZinc optimization instances; alternatively, FlatZinc could be extended in order to directly support these extension.

Furthermore, our proposal has built-in support for meta-constraints, covering all kinds described in [13]. Meta-constraints can be very helpful in the modeling process, since they allow us to abstract to a higher level. For instance, one can impose certain priority among a set of soft constraints, instead of having to define a concrete weight for each of them.

Concerning the possible underlying solving methods, in a different work [1] we consider the use of SMT (as has been done for general CSPs in [2, 3]) and weighted-SMT.

The rest of the paper is structured as follows. In Section 2 we recall some basic concepts on weighted CSP. Section 3 is the main section of the paper, being devoted to our proposal of extension of MiniZinc in order to support WCSP and meta-constraints, and to the discussion of several ways of solving the resulting W-MiniZinc instances. Some conclusions are given in Section 4.

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 [6, 10, 9] 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

[3] Although the original Zinc [15] specification had annotations for soft constraints, these were never implemented.

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 (WCSP) [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 *is a set of variables* $\{x_1, \ldots, x_n\}$*,* D *is a set of domains* $\{d(x_1), \ldots, d(x_n)\}$ *containing the values the variables may take, and* C *is a set of constraints* $\{C_1, \ldots, C_m\}$*. 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 *is a subset of* X*.*

An (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 their 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 W-MiniZinc

Many existing WCSP solving systems and languages consider an extensional approach, i.e., deal with instances consisting of an enumeration of good/nogood tuples for hard constraints and of 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.

Consider for instance two variables x and y, with domain 1..2, and a soft constraint $x < y$ whose falsification cost is 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 W-MiniZinc we would express it as: *constraint* $x < y$ @ $\{1\}$.

Simple weighted constraints The most basic type of weighted constraint proposed in W-MiniZinc is of the form:

⟨*w-constraint-item*⟩ ::= `constraint` ⟨*expression*⟩ @ {⟨*expression*⟩};

where the first expression is a MiniZinc expression defining a constraint and the second one is an integer arithmetic expression whose value is the weight (cost) of falsifying the constraint.The weight expression can either be evaluable in compilation time, or contain decision variables. For example,

```
constraint a < b @ {10};
```

or

```
constraint a < b @ {a - b + 1};
```

Labeled weighted constraints Labels can be used to refer to a constraint inside another constraint. They are also essential for the definition of meta-constraints.

⟨*w-constraint-item*⟩ ::= `constraint` #*label*: ⟨*expression*⟩ @ {⟨*expression*⟩};

For example,

```
constraint #A: a > b @ {1};
constraint #B: a > c @ {2};
constraint #C: a > d @ {1};
constraint not A /\ not B -> C;
```

Notice that labels introduce aliasing on constraints and allow users to state things like:

```
constraint #A: a > b @ {10};
constraint A @ {2};
constraint A @ {3};
```

which should be equivalent to:

```
constraint a > b @ {10};
constraint a > b @ {2};
constraint a > b @ {3};
```

meaning that the cost of falsifying `a > b` is 15.

Notice also that we only allow the use of weights in *w-constraint-item*s. This limitation has the drawback of not being portable to *user defined predicates*. Therefore it is worthy to study the possibility of allowing the use of weights attached to constraint expressions in predicate definitions too.

Meta-constraints In order to provide a higher level of abstraction in the modeling of over-constrained problems, in [13] several meta-constraints[4] are proposed. To cover them a third kind of weighted constraint is necessary:

$$\langle \textit{w-constraint-item} \rangle ::= \texttt{constraint}\ \#\textit{label}\texttt{:}\ \langle \textit{expression} \rangle\ \texttt{@}\ \{\_\};$$

where the underscore "_" denotes an undefined weight. The value of this undefined weight is computed at compilation 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:

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 whilst worker 2 *should not* perform it, then it is better to violate the first constraint than the second. In our proposal the following meta-constraints refer to this aspect:
   - `samePriority(`*List*`)`, where *List* is a list[5] 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 more 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. This can be easily stated in W-MiniZinc, since decision variables can be used in the expressions defining the weights. For instance, the previous soft constraint could be specified as follows:

```
constraint worked_turns < 6 @ {base_cost + worked_turns - 6};
```

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

[5] In several of these meta-constraints, instead of using lists, we could use sets.

3. **Homogeneity.** Sometimes, the degree of violation 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 propose 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 (among other things) 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:

   ```
   constraint #A: not_last_turn_day_1 @ {w1};
   constraint #B: not_first_turn_day_2 @ {w2};
   constraint (not A) -> B;
   ```

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

   ```
   constraint #A: not_last_turn_monday @ {w1};
   constraint #B: not_last_turn_tuesday @ {w2};
   constraint (not A /\ not B) -> NewConstraint;
   ```

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

The meta-constraint predicates (`samePriority`, `priority`, `atLeast` and `homogeneous`) must be preceded by the `metaConstraint` item.

3.1 Translation of meta-constraints

Apart from the possibility of letting the solvers directly deal with (some) meta-constraints, as it can be done for global constraints in MiniZinc, our approach

for managing meta-constraints consists on removing all of them by reformulating them into W-MiniZinc constraints.

In order to deal with the *priority* (1) meta-constraints, we can create a system of linear inequations on the undefined weights of the soft constraints referenced by the meta-constraint. The inequations will be 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:

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

the following set of inequations could be 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 must be solved in compilation time, so that a model, i.e., a solution for the undefined weights satisfying the inequations can be found. Following the previous example, a solution could be, e.g.,:

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

This would allow to reformulate the original instance into an equivalent W-MiniZinc instance without undefined weights:

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

Hence, with the meta-language 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 compilation time.

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

In order to deal with the *violation degree* (2) meta-constraints, we can create an auxiliary integer variable for each weight expression. These variables must be restricted to be equal to zero if the constraint is satisfied, and equal to the weight expression if the constraint is falsified.

Finally, we can deal with the *homogeneity* (3) 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*`)` could be reformulated into:

```
constraint count(ListReif,1,n);
constraint n >= p*len;
```

where `ListReif` is the list of 0/1 integer variables resulting from reifying the list of constraints referenced in *List*, and `len` is the length of *List*. Recall that `count(`l`,`e`,`n`)` is the MiniZinc 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 `homogeneous(`*ListOfLists*`,`p`)` can be reformulated into:

```
constraint count(ListOfLists[1],1,n[1]);
constraint count(ListOfLists[2],1,n[2]);
...
constraint abs(n[1]/len[1] - n[2]/len[2]) <= p;
constraint abs(n[1]/len[1] - n[3]/len[3]) <= p;
...
constraint abs(n[l-1]/len[l-1] - n[l]/len[l]) <= p;
```

where `l` is the length of *ListOfLists*, and `len`$[i]$ is the length of the i-th list in *ListOfLists*.

3.2 Solving

In order to solve a W-MiniZinc instance we consider two possibilities:

1. Translating the W-MiniZinc instance into a minimization instance in MiniZinc. In this possibility we can create a fresh integer variable o_i for each soft constraint C_i @ $\{w_i\}$. This variable must either evaluate to the weight w_i if the constraint C_i is violated, or to 0 otherwise. Therefore, for each W-MiniZinc soft clause we can generate the following constraints:

   $$o_i \geq 0 \qquad c_i \vee (o_i > 0) \qquad (o_i = w_i) \vee (o_i = 0)$$

   Notice that the first clause forbids negative weights (this is convenient since w_i can be an expression containing decision variables). The second and third clauses state that if c_i is falsified, then the cost is w_i, and otherwise it is 0. Since we are minimizing costs it will never happen that c_i is satisfied and the cost o_i is greater than 0.
   Finally, we need to introduce an integer variable that represents the sum of the o_i variables:

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

   Then the goal is minimizing O.
2. Modifying the FlatZinc language by allowing weights on its constraints too, in what would be called W-FlatZinc. We should use basic MiniZinc to FlatZinc translation and then associate the corresponding weight to the whole translation of the original weighted constraint.

The solving approach, i.e., whether the W-MiniZinc instance must be translated into an optimization FlatZinc instance or into a W-FlatZinc one, should be specified by a command line switch of the MiniZinc compiler.

Although optimization problems with weighted constraints could be solved as multilevel optimization problems, for the sake of clarity we propose forbidding them in W-MiniZinc.

3.3 Syntax alternatives

It is possible to avoid the use of labels by associating *violation variables* (with domain {0,1}) to weighted constraint expressions, as a sort of reification. Then, the objective would be to minimize the cost expression consisting on the sum of the products of these violation variables by the cost of the corresponding weighted constraints. However, notice that this alternative does not consider the generation of any sort of W-FlatZinc.

Alternatively, we could also use the *annotations* facilities of MiniZinc to avoid extending too much the grammar. This alternative suggests a natural extension to a possible resulting W-FlatZinc language, since annotations can also be used in FlatZinc. Nevertheless, with the use of annotations the sweetness of our (syntactically sugared) proposal would be lost.

4 Conclusions

The MiniZinc extension we have presented, W-MiniZinc, covers the lack of a high-level language with declarative facilities for over-constrained problems. W-MiniZinc allows the intensional description of over-constrained problems, and includes some meta-constraints, which increase the capability to easily model several real problems.

As further improvements we could consider other meta-constraints, such as for describing multilevel preferences. We should also study how to deal with weighted user-defined predicates and with soft global constraints.

Finally, we want to thank the comments and suggestions of the anonymous referees.

References

1. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. A Proposal for Solving Weighted CSP with SMT. In *ModRef 2011*, Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation, 2011 (to appear).
2. 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*, Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation, pages 30–44, 2009.

3. M. Bofill, J. Suy, and M. Villaret. A System for Solving Constraint Satisfaction Problems with SMT. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT'10*, volume 6175 of *LNCS*, pages 300–305. Springer, 2010.
4. M. C. Cooper, S. De Givry, and T. Schiex. Optimal soft arc consistency. In *IJCAI 2007*, Proceedings of the 20th International Joint Conference on Artificial Intelligence, 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*, Proceedings of the 19th International Joint Conference on Artificial Intelligence, pages 84–89, 2005.
6. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21 – 70, 1992.
7. 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.
8. I. Gent, I. Miguel, and A. Rendl. Optimising Quantified Expressions in Constraint Models. In *ModRef 2010*, Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation, 2010.
9. J. Larrosa and P. Meseguer. Partition-Based Lower Bound for Max-CSP. *Constraints*, 7:407–419, 2002.
10. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149 – 163, 1999.
11. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-Consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.
12. 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.
13. T. Petit, J. C. Regin, and C. Bessiere. Meta-constraints on violations for over constrained problems. In *ICTAI 2000*, Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence, pages 358–365, 2000.
14. O. Roussel and C. Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *Computing Research Repository*, 2009.
15. P. Stuckey, M. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The g12 project: Mapping solver independent models to efficient solutions. In *CP 2005*, volume 3709 of *LNCS*, pages 13–16. Springer, 2005.
16. W. J. van Hoeve. Over-Constrained Problems. In M. Milano and P. V. Hentenryck, editors, *Hybrid Optimization: the 10 years of CPAIOR*, volume 45 of *Springer Optimization and Its Applications*, chapter 6, pages 191–226. Springer, 2011.