Automated Symmetry Breaking and Model Selection in CONJURE

Ozgur Akgun[1], Alan M. Frisch[2], Ian P. Gent[1], Bilal Syed Hussain[1], Christopher Jefferson[1], Lars Kotthoff[3], Ian Miguel[1], and Peter Nightingale[1]

[1] University of St Andrews
[2] University of York
[3] Cork Constraint Computation Centre

Abstract Constraint modelling is widely recognised as a key bottleneck in applying constraint solving to a problem of interest. The CONJURE automated constraint modelling system addresses this problem by automatically refining constraint models from problem specifications written in the ESSENCE language. ESSENCE provides familiar mathematical concepts like sets, functions and relations nested to any depth. To date, CONJURE has been able to produce a set of alternative model kernels (i.e. without advanced features such as symmetry breaking or implied constraints) for a given specification. The first contribution of this paper is a method by which CONJURE can break symmetry in a model as it is introduced by the modelling process. This works at the problem class level, rather than just individual instances, and does not require an expensive detection step after the model has been formulated. This allows CONJURE to produce a higher quality set of models. A further limitation of CONJURE has been the lack of a mechanism to select among the models it produces. The second contribution of this paper is to present two such mechanisms, allowing effective models to be chosen automatically.

1 Introduction and Background

For constraint programming to achieve its potential widespread industrial and academic use, reducing the *modelling bottleneck* [29] is of central importance. This is the problem of formulating a problem of interest as a constraint model suitable for input to a constraint solver. There are typically many possible models for a given problem, and the model chosen can dramatically affect the efficiency of constraint solving. This presents a serious obstacle for non-expert users, who have difficulty in formulating a good (or even correct) model from among the many possible alternatives. Therefore, automating constraint modelling is a desirable goal. Numerous approaches have been taken to automate aspects of constraint modelling, including: case-based reasoning [23]; theorem proving [6]; automated transformation of medium-level solver-independent constraint models [27, 28, 30, 33]; and refinement of abstract constraint specifications [9] in languages such as ESRA [8], ESSENCE [10], $\mathcal{F}$ [18] or Zinc [21, 25]. Some systems [2–4, 7, 22] aim to learn constraint models from positive or negative examples.

This paper focuses on the refinement-based approach, in which a user writes abstract constraint specifications to describe a problem at a higher level than that where

modelling decisions are normally made. Abstract constraint specification languages, e.g. ESSENCE and Zinc, support abstract variables with types for common mathematical structures such as sets, multisets, functions, and relations, as well as nested types, such as set of sets and multiset of functions. Problems can often be specified very concisely in this way. For example, the Social Golfers Problem [17], which is to find a set of partitions of golfers subject to some constraints, can be specified directly (see Fig. 1) without the need to model the sets or partitions as matrices of Integer variables.

We use ESSENCE in this paper [10]. An ESSENCE specification, such as that in Fig. 1, identifies: the input parameters of the problem class (`given`), whose values define an instance; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy (`such that`). An objective function may also be specified (`min/maximising`) and identifiers may be declared (`letting`). Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. Our CONJURE system[4] [1] uses refinement rules to convert an ESSENCE specification into the solver-independent constraint modelling language ESSENCE′ [30]. From ESSENCE′ we use SAVILE ROW[5] to translate the model into input for a particular constraint solver while performing solver-specific model optimisations.

CONJURE has been able to produce the *kernels* of constraint models, without advanced features like symmetry breaking often used by experts to improve model performance. The first contribution of this paper is to automate the generation of symmetry-breaking constraints. Much symmetry enters constraint models through the process of constraint modelling [11]. CONJURE exploits this by breaking symmetry as it enters the model. This obviates the need for an expensive symmetry detection step following model formulation, as used by other approaches [24,26]. The added symmetry breaking constraints hold for the entire parameterised problem class — not just a single problem instance — without the need to identify graph automorphisms.

The second contribution of this paper is to automate model selection. Previously, CONJURE has been able to produce a (typically large) set of alternative models through the application of alternative refinement rules, but not to select among these models. CONJURE can now automatically select the best models for a problem class.

2 Automated symmetry breaking

Symmetry enters constraint models in two ways. Some problems have inherent symmetries, which if not broken get reflected in the model. Other symmetries are introduced by the modelling process; in this case a single solution to the problem corresponds to multiple assignments to the variables of the model. We call these *model* symmetries. As an example, consider the Social Golfers Problem (Fig. 1), which requires finding a set of w partitions. If this set is modelled as an array indexed by $1..w$ then all $w!$ permutations of the array correspond to the same set. This symmetry is introduced when an arbitrary decision is made about which set element goes in which cell of the array. Similarly, if the $g*s$ *Golfers* are modelled by the Integers $1..g*s$ then $g*s$ symmetries are introduced because of the arbitrary decision of which golfer corresponds to which

[4] `https://bitbucket.org/stacs_cp/conjure-public/`
[5] `http://savilerow.cs.st-andrews.ac.uk`

```
given   w, g, s : int(1..)
letting Golfers be new type of size g * s
find    sched   : set (size w) of partition (regular, size g)
                  from Golfers
such that
 forAll week1, week2 in sched, week1 != week2 .
   forAll group1 in parts(week1) .
     forAll group2 in parts(week2) .
       |group1 intersect group2| < 2
```

Figure 1: ESSENCE specification of the Social Golfers Problem.

Integer. The problem-specification language ESSENCE has been designed such that, unlike other modelling languages, problems can be specified without having to make the arbitrary decisions that introduce model symmetries.

Frisch et al. [11] show how each modelling rule of CONJURE can be extended to generate a description of the symmetries it introduces and how the generated descriptions can be composed to form a description of the symmetries introduced into the model. The intention was that this could then be used to generate symmetry-breaking constraints, though these descriptions were never fully developed into a method for automatically generating symmetry-breaking constraints.

The current version of CONJURE takes a different approach to generating symmetry breaking constraints: every rule that introduces symmetries also generates a constraint to break those symmetries. CONJURE has 28 such rules. There is only one rule which does not break all symmetries it introduces – the rule that refines an unnamed type, such as $Golfers$, to a range of Integers. This is because each unnamed type can be used in multiple places, and the symmetry of an unnamed type must be broken in a globally consistent way. All the other symmetries we introduce are independent, so we can add constraints which immediately break each introduced group of symmetries in a valid and complete manner. This leads to globally valid and complete symmetry breaking. We plan to handle unnamed types in the future.

To illustrate how CONJURE rules can be extended to generate symmetry-breaking constraints, consider the rule given below to build the explicit representation of a set.

```
Representation: Set~Explicit~Sym
Matches:        set (size &n, ..) of &tau
Produces:       refn : matrix indexed by [int(1..&n)] of &tau
Constraint:     allDiff(refn)
```

This rule transforms a set of a size n into a matrix with n index values, where each value in the matrix is a member of the set. A constraint is imposed to ensure that the cells of the matrix are all different. For any `tau` other than Integers or Booleans, CONJURE has to further decompose the `allDiff` constraint into $O(n^2)$ not-equal constraints.

Now consider extending this rule to generate a constraint to break the symmetry it introduces, that the index values of the matrix can be permuted in any way. The simplest way to break this symmetry is to impose a total order on the elements of the matrix. As the elements of the matrix can be any type `tau` we introduce two new operators, $\dot{\leq}$ and $\dot{<}$. These operators provide a total ordering (and a strict version of the same total ordering) for all types in CONJURE. These orderings are not intended to be

"natural" and are not available to ESSENCE users. They are used only in refinement rules to generate effective symmetry-breaking constraints. Using these orderings, the `Set~Explicit~Sym` rule is modified to a rule that breaks all the symmetries it introduces:

```
Representation: Set~Explicit
Matches:        set (size &n, ..) of &tau
Produces:       refn : matrix indexed by [int(1..&n)] of &tau
Constraint:     forAll i : int(1..&n-1) . refn[i] .< refn[i+1]
```

Rather than introducing a chain of $\dot{\leq}$ constraints, this rule exploits the fact that the elements of the set are required to be all different and strengthens the ordering to $\dot{<}$ constraint. This replaces $O(n^2)$ not-equal constraints with only $O(n)$ $\dot{<}$ constraints.

Other refinement rules can exploit the fact that symmetry breaking is performed immediately to produce more efficient refinements. Consider refining the constraint $S = T$ by representing the sets S and T as matrices S' and T' with the `Set~Explicit~Sym` representation. To find if S' and T' represent the same set we must check if each element of S' is equal to *any* element of T' and whether the two sets have the same cardinality, since the order of elements in the matrices can be different. However, when the `Set~Explicit` representation is used we can refine $S = T$ to the constraint $S' = T'$, because each assignment of S is represented by exactly one assignment to S', which satisfies the symmetry breaking constraint. This gives a much smaller constraint, which propagates much more efficiently.

We illustrate the new approach to symmetry-breaking by showing how the SGP specification (Fig. 1) is refined into a model with symmetry-breaking constraints. We consider generating only one model. We will consider only how the decision variables are refined, ignoring all constraints other than symmetry-breaking constraints. First, CONJURE replaces `type of size g*s` with `int(1..g*s)`:

```
given  w, g, s : int(1..)
find   sched'  : set (size w) of partition (regular, size g) from int(1..g*s)
```

After this, CONJURE refines the type of the decision variable by rewriting the outer `set` constructor using the `Set~Explicit` rule given in the previous section. This generates the following refinement.

```
given       w, g, s : int(1..)
find        sched'  : matrix indexed by [int(1..w)] of
                          partition (regular, size g) from int(1..g*s)
such that   forAll i : int(1..w-1). sched'[i] .< sched'[i+1]
```

This refinement step shows all of the important features of our method. CONJURE has introduced a new, compact constraint which both breaks symmetry, and ensures all members of the matrix are distinct. Next, it transforms the partition into a set of sets:

```
given       w, g, s : int(1..)
find        sched'' : matrix indexed by [int(1..w)] of
                          set (size g) of set (size (g*s)/g) of int(1..g*s)
such that   forAll i : int(1..w-1). sched''[i] .< sched''[i+1],
            forAll j : int(1..w).
                forAll k1,k2 in sched''[j], k1 != k2. | k1 intersect k2 | = 0
```

This refinement does not appear to have changed the symmetry-breaking constraint but it has in fact been refined from a partition to a set of sets. CONJURE has also added a constraint to impose that the cells of the partition are distinct. This structural constraint constrains the sets to be disjoint. CONJURE now applies `Set~Explicit` again.

```
given       w, g, s : int(1..)
find        sched''' : matrix indexed by [int(1..w), int(1..g)]
                 of set (size (g*s)/g) of int(1..g*s)
such that   forAll i : int(1..w-1). sched'''[i,..] .< sched'''[i+1,..],
            forAll j : int(1..w). forAll k : int(1..g-1).
                sched'''[j,k] .< sched'''[j,k+1]
```

The first constraint here is the refined version of the already existing symmetry-breaking constraint. Once again by design the $\dot{<}$ constraint maps naturally to the matrices used in refinement. The second constraint is the symmetry breaking on matrix of sets, now transformed into a matrix of matrices. CONJURE uses the same refinement rule, even though we are now refining a set inside a matrix. CONJURE automatically deals with the array indices and inserts the outer `forAll j : int(1..w)` in a process called *lifting*. We finally apply `Set~Explicit` once more and change the $\dot{<}$ and $\dot{\leq}$ constraints to their final form – lexicographic ordering constraints on matrices and ordering on Integers.

```
given       w, g, s : int(1..)
find        sched''' : matrix indexed by [int(1..w), int(1..g),int(1..(g*s/g))]
                 of int(1..g*s)
such that   forAll i : int(1..w-1). sched'''[i,..,..] <lex sched'''[i+1,..,..],
            forAll j : int(1..w). forAll k : int(1..(g*s)/g-1).
               sched'''[j,k,..] <lex sched'''[j,k+1,..]
            forAll j : int(1..w). forAll k : int(1..g)
                forAll l : int(1..(g*s)/g-1). sched'''[j,k,l] < sched'''[j,k,l+1]
```

If CONJURE had not the broken symmetries immediately, but instead used the `Set~Explicit~Sym` representation, the constraints requiring each partition in the outermost set to be different would now be very complex. This shows the benefit of breaking symmetries as soon as they are introduced, rather than delaying and using a general technique for symmetry breaking after model generation is finished.

3 Automated Model Selection

Our previous work [1] shows that CONJURE can successfully refine a set of model kernels (i.e. excluding symmetry breaking and implied constraints) from a given specification, and that this set contains the kernels of *effective* models. However, without symmetry breaking the performance of these model kernels is poor since refinement of abstract types naturally introduces a great deal of symmetry. Therefore, the symmetry breaking approach described above is a necessary step in producing practically useful models. Having thus enhanced CONJURE the natural next step is to provide a means to select an effective model automatically. We propose and evaluate two such approaches: a lightweight heuristic based purely on an analysis of model structure and an approach that uses a set of training instances to perform model selection by means of a race.

3.1 The Compact heuristic

If time is limited it is sensible to provide a rapid model selection method, avoiding both generating all models and training using instance data. Our solution is a heuristic employed during refinement to commit greedily to promising modelling choices at each point where an abstract type or a constraint expression may be refined in multiple ways.

It is named Compact since it favours transformations that produce smaller expressions. For an abstract type, we define an ordering as follows: concrete domains (such as `bool`, `matrix`) are smaller than abstract domains; within concrete domains, `bool` is smaller than `int` and `int` is smaller than `matrix`. These rules are applied recursively, so that a one-dimensional matrix of `int` is smaller than any two-dimensional matrix. Abstract type constructors have the ordering `set < mset < function < relation < partition`, which is also applied recursively. Compact will select the smallest domain according to this order. For a constraint expression (and the objective), Compact chooses the refinement with the most shallow abstract syntax tree.

3.2 Racing

Our second selection method takes as input a set of instances representative of the distribution of instances a user wishes to solve. Our measure of quality of a model with respect to an instance is the time taken for SAVILE ROW to instantiate the model and translate for input to the MINION constraint solver [13] plus the time taken for MINION to solve the instance. We include the time taken by SAVILE ROW since it adds desirable instance-specific optimisations to the model, such as common subexpression elimination [14]. Given a parameter $\rho \geq 1$, a model is *ρ-dominated* on an instance by another model if the measure for the second model is at least ρ times faster than the first.

We iterate over the set of instances and conduct a *race* [5] for each. The set of models entered into the race for instance i are the winners of the race for instance $i-1$, with all models entered in the first race. The 'winners' of an instance race are the models not ρ-dominated by any other model. After we have iterated over all of the supplied instances, the subset of models remaining is selected for the specified class. This naturally suggests the notion of a *model portfolio*, analogous to algorithm portfolios [16, 19].

A set of instances is *ρ-fractured* if every model is ρ-dominated on at least one instance. If the supplied set of instances is fractured, races run with different instance orderings can produce disjoint sets of models. We observe this experimentally in Section 3.4 and discuss its consequences.

3.3 Case Study: Equidistant Frequency Permutation Arrays

We illustrate the model selection process using the Equidistant Frequency Permutation Array (EFPA) problem [20]: 'The problem has parameters v, q, λ, d and it is to find a set E of size v, of sequences of length $q\lambda$, such that each sequence contains λ of each symbol in the set $\{1, \ldots, q\}$. For each pair of sequences in E, the pair are Hamming distance d apart (i.e. there are d places where the sequences disagree)'.

This problem is specified in ESSENCE (see Fig. 2) with a single abstract decision variable E and two constraints. The first ensures that each codeword must contain each symbol λ times, the second that each pair of codewords must differ in exactly d places. CONJURE refines this specification into 45 models. The type of E is a fixed size set of total functions. The outer set is always modelled using the explicit representation (as a vector of the inner type) and the symmetry is broken by constraining the elements of the vector to be in increasing order according to $\dot{<}$. The total function is refined in two ways: to a vector, or to a relation. In the latter case the relation is refined in four different

```
given       d, lambda, q, v : int(1..)
letting     Character     be domain int(1..q)
letting     Index         be domain int(1..lambda * q)
letting     String        be domain function (total)
                             Index --> Character
find        E           : set (size v) of String
such that   forAll s in E . forAll a : Character .
                 (sum i : Index . toInt(s(i) = a)) = lambda,
            forAll s1, s2 in E, s1 != s2 .
                 (sum i : Index . toInt(s1(i) != s2(i))) = d
```

Figure 2: ESSENCE specification of the EFPA Problem.

ways, giving five representations of E in total. Subsets of these five are channelled and constraints are stated on different representations to create 45 models.

For EFPA we use 24 instances from Huczynska et al. [20], and 12 easier instances that were created by taking the satisfiable instances from Huczynska et al. and reducing v by one. Identifying instances by the tuple $\langle d, \lambda, q, v\rangle$, the first instance we race is $\langle 3, 7, 7, 5\rangle$. This instance is exceptionally discriminating. The number of winners is 4, so we have eliminated 41 models at this stage. We will see in Section 3.4 that not all problems converge so quickly. Second, the remaining models are raced on the instance $\langle 3, 8, 8, 6\rangle$. This does not eliminate any models, although they are ranked in a different order. This process is continued for another 30 instances that eliminate no models. Instance $\langle 6, 4, 3, 12\rangle$ eliminates one model, leaving three. Finally, the last three instances eliminate no more models so the final winning set has three models.

All of the final set of models contain the vector representation of the total function. Two of the models refine the function to a relation, then to a two-dimensional matrix of Boolean variables (which is channelled with the vector). These two models differ in one constraint. The relative similarity of these three models shows that on this problem there is a clear cluster of similar winners among a more diverse set of models.

For this problem, Compact generates the model which uses the vector representation for the function variable without any channelling. Although it uses far less information and is very quick in comparison to racing, it manages to find one of the 'winner' models.

3.4 Experimental Evaluation

In this section, we present the results of model selection for the five problem classes presented in Table 1: EFPA [20], Social Golfers Problem (SGP) [15], Progressive Party Problem (PPP) [32], the SONET network design problem [31], and Error Correcting Codes (ECC) [12]. Although not generally feasible in practice, for the purpose of this experiment we ran a race for every model on every instance with no pruning of models between races. We set $\rho = 2$ and a timeout of one hour. Furthermore, a model that solves an instance within ten seconds is considered to be non-dominated on that instance. The results presented in the *Winner set size* column of Table 1 show the number of non-dominated models in each case. For the second problem class, SGP, the set of instances is fractured; every winner set contains either model 2 or model 3 but not both.

Now consider the performance of the racing scheme. Notice that the winner set of a race must contain all the non-dominated models. It may contain further models that

Inputs			Steps to convergence		Results	
Problem	Models	Instances	Mean	Std. Dev.	Winner set size	Compact
EFPA	45	36	9.64	4.65	1	Yes
SGP	4	37	5.14	1.78	Fractured	Yes
PPP	81	11	5.67	1.47	4	No
ECC	108	26	2.75	0.43	4	No
SONET	27	47	4.30	1.93	1	Yes

Table 1: Experimental results.

were not eliminated because of the eager pruning policy. Such models are dominated on some instance by models that were eliminated earlier in the racing process. The number and identity of these extra models is dependent on the order that the instances are considered. Also dependent on this order is the rate of convergence.

In order to test the importance of instance order, we ran 50 races with randomly-selected instance orders. The racing scheme does not know if the problem instances are fractured, though in some cases it may detect that it is. We make the distinction solely for the sake of this study. For the four non-fractured problem classes, all 50 sample races yielded a winner set comprising exactly the non-dominated models. In contrast, the SGP does exhibit fracturing. On the 50 runs every winner set is a singleton comprising either model 1, 2 or 3. The mean and standard deviation of the number of instances raced before reaching the final model set are given in Table 1 under the *Steps to convergence* heading.

Table 1 also presents whether Compact manages to generate a model that is in one of the winner sets found by racing. It finds a winner model for two out of four non-fractured problem classes. Moreover, it finds a winner model for one of the subdivisions in the fractured class, SGP. This is a promising result, considering that Compact works with far less information than racing and is very cheap.

4 Conclusions

This paper has demonstrated significant progress towards the goal of automated constraint modelling. We have shown how symmetry can be broken cheaply and automatically as it enters the model through the modelling process, increasing the quality of the models that CONJURE can produce beyond model kernels. Furthermore, we have shown how CONJURE can select *effective* models using a racing process and the Compact heuristic.

Acknowledgements We thank the anonymous reviewers for their comments. This research is supported by UK EPSRC grants no EP/H004092/1 and EP/K015745/1 and EU FP7 grant 284715.

References

1. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAAI-11: Twenty-Fifth Conference on Artificial Intelligence (2011)
2. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: 18th International Conference on Principles and Practice of Constraint Programming. pp. 141–157 (2012)
3. Bessiere, C., Coletta, R., Freuder, E.C., O'Sullivan, B.: Leveraging the learning power of examples in automated constraint acquisition. In: 10th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 3258, pp. 123–137. Springer Berlin Heidelberg (2004)
4. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: Acquiring constraint networks using a SAT-based version space algorithm. In: AAAI 2006. pp. 1565–1568 (2006)
5. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 11–18. Morgan Kaufmann (2002)
6. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: Proc. of ECAI 2006. pp. 73–77. IOS Press (2006)
7. Coletta, R., Bessiere, C., O'Sullivan, B., Freuder, E.C., O'Connell, S., Quinqueton, J.: Semi-automatic modeling by constraint acquisition. In: 9th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 2833, pp. 812–816. Springer Berlin Heidelberg (2003)
8. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: LOPSTR 2003. pp. 214–232 (2003)
9. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: Proc. of the IJCAI 2005. pp. 109–116 (2005)
10. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3) pp. 268–306 (2008), http://dx.doi.org/10.1007/s10601-008-9047-y
11. Frisch, A.M., Jefferson, C., Martinez-Hernandez, B., Miguel, I.: Symmetry in the generation of constraint models. In: Proceedings of the International Symmetry Conference (2007)
12. Frisch, A., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003, Lecture Notes in Computer Science, vol. 2833, pp. 318–332. Springer Berlin Heidelberg (2003)
13. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings ECAI 2006. pp. 98–102 (2006)
14. Gent, I.P., Miguel, I., Rendl, A.: Common subexpression elimination in automated constraint modelling. In: Workshop on Modeling and Solving Problems with Constraints. pp. 24–30 (2008)
15. Gent, I.P., Walsh, T.: CSPlib: a benchmark library for constraints. Tech. rep. (Apr 16 1999), http://citeseer.ist.psu.edu/5054.html; http://www.cs.strath.ac.uk/ apes/reports/apes-09-1999.ps.gz
16. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
17. Harvey, W.: Symmetry breaking and the social golfer problem. In: Proc. SymCon-01: Symmetry in Constraints, co-located with CP 2001. pp. 9–16 (2001)
18. Hnich, B.: Thesis: Function variables for constraint programming. AI Commun 16(2), 131–132 (2003)

19. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
20. Huczynska, S., McKay, P., Miguel, I., Nightingale, P.: Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In: Proc. CP 2009. pp. 50–64 (2009)
21. Koninck, L.D., Brand, S., Stuckey, P.J.: Data independent type reduction for zinc. In: ModRef10 (2010)
22. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI). vol. 1, pp. 45–52 (2010)
23. Little, J., Gebruers, C., Bridge, D.G., Freuder, E.C.: Using case-based reasoning to write constraint programs. In: CP. p. 983 (2003)
24. Mancini, T., Cadoli, M.: Detecting and breaking symmetries by reasoning on problem specifications. In: Abstraction, Reformulation and Approximation. Lecture Notes in Computer Science, vol. 3607, pp. 165–181. Springer Berlin Heidelberg (2005)
25. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the zinc modelling language. Constraints 13(3) (2008), http://dx.doi.org/10.1007/s10601-008-9041-4
26. Mears, C., Niven, T., Jackson, M., Wallace, M.: Proving symmetries by model transformation. In: 17th International Conference on Principles and Practice of Constraint Programming. pp. 591–605. CP'11, Springer-Verlag, Berlin, Heidelberg (2011)
27. Mills, P., Tsang, E., Williams, R., Ford, J., Borrett, J.: EaCL 1.5: An easy abstract constraint optimisation programming language. Tech. rep., University of Essex, Colchester, UK (December 1999)
28. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack., G.: Minizinc: Towards a standard CP modelling language. In: Proc. of CP 2007. pp. 529–543 (2007)
29. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: Principles and Practice of Constraint Programming - CP 2004. pp. 5–8 (2004)
30. Rendl, A.: Thesis: Effective Compilation of Constraint Models. Ph.D. thesis, University of St. Andrews (2010)
31. Smith, B.M.: Search strategies for optimization: Modelling the sonet problem. Tech. Rep. Research Report APES-70-2003 (2003), presented at 2nd International Workshop on Reformulating Constraint Satisfaction Problems
32. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: Integer linear programming and constraint programming compared. In: CP '95. pp. 36–52 (1995)
33. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA (1999)