Discriminating Instance Generation for Automated Constraint Model Selection

Ian P. Gent[1], Bilal Syed Hussain[1], Christopher Jefferson[1], Lars Kotthoff[2], Ian Miguel[1], Glenna F. Nightingale[1], and Peter Nightingale[1]

[1] School of Computer Science, University of St Andrews, UK
{ian.gent, bh246, caj21, ijm, gfe2, pwn1}@st-andrews.ac.uk
[2] INSIGHT Centre for Data Analytics, Ireland, larsko@4c.ucc.ie

Abstract One approach to automated constraint modelling is to generate, and then select from, a set of candidate models. This method is used by the automated modelling system CONJURE. To select a preferred model or set of models for a problem class from the candidates CONJURE produces, we use a set of training instances drawn from the target class. It is important that the training instances are discriminating. If all models solve a given instance in a trivial amount of time, or if no models solve it in the time available, then the instance is not useful for model selection. This paper addresses the task of generating small sets of discriminating training instances automatically. The instance space is determined by the parameters of the associated problem class. We develop a number of methods of finding parameter configurations that give discriminating training instances, some of them leveraging existing parameter-tuning techniques. Our experimental results confirm the success of our approach in reducing a large set of input models to a small set that we can expect to perform well for the given problem class.

1 Introduction and Background

Numerous approaches have been taken to automating aspects of constraint modelling, including: learning models from examples [8,4,5,17,3]; automated transformation of medium-level solver-independent constraint models [23,21,24,20]; theorem proving [7]; case-based reasoning [18]; and refinement of abstract constraint specifications [10] in languages such as ESRA [9], ESSENCE [11], $\mathcal{F}$ [14] or Zinc [19,16]. We focus on the refinement approach, where a user writes a constraint specification describing a problem above the level of abstraction at which modelling decisions are made. Constraint specification languages support abstract decision variables with types such as set, multiset, relation and function, as well as *nested* types, such as set of sets and multiset of relations. Therefore, problems can typically be specified very concisely. However, existing constraint solvers do not support these abstract decision variables directly, so abstract constraint specifications must be refined into concrete constraint models.

We use ESSENCE [11]. An ESSENCE specification (see Fig. 1) identifies: the 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 declared (`letting`). Our CONJURE system[3] [2] employs refinement rules to convert

[3] `http://bitbucket.org/stacs_cp/conjure-public`

an ESSENCE specification into the solver-independent constraint modelling language ESSENCE′ [23]. We use SAVILEROW[4] [22] to translate an ESSENCE′ model into input for a particular constraint solver while performing solver-specific model optimisations.

By following alternative refinement paths CONJURE typically produces a large set of models for a given ESSENCE specification. In our previous work [1] we developed a racing process to select among these candidate models, in which a set of training instances drawn from the problem class being modelled is used to gauge relative model performance. For this process to be effective, it is important that the instances chosen are *discriminating*: if all models solve a given instance in a trivial amount of time, or if no model solves it in the time available, then the instance is not useful for model selection. Our previous work assumed that such instances were given. In this paper we address the task of generating discriminating training instances automatically.

2 Racing for Automated Model Selection

Our approach to model selection follows that we reported in [1], but with an improved method of producing a set of winning models described below. It takes as input a set of instances drawn from the target problem class. Our performance measure of a model with respect to an instance is the time taken for SAVILEROW to instantiate the model and translate for input to the MINION constraint solver [12] plus the time taken for MINION to solve the instance. We include the time taken by SAVILEROW since it adds desirable instance-specific optimisations, such as common subexpression elimination [13].

We conduct a *race* [6] for each provided instance. 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. The 'winners' of an instance race are the models not ρ-dominated by any other model. So that trivial instances do not discriminate we do not consider any model that solves within 1s to be dominated. All models enter each race, but for efficiency a model is terminated as soon as it is $\rho-$dominated by some other model. Furthermore, the order in which the models are executed is influenced by their performance in previous races: well-performing models are executed first to establish a good ρ bound early. In order to guide our exploration of the instance space we assign a discriminatory quality value to an instance with respect to the results of the race run. This is the fraction of models that are ρ-dominated.

A set of instances is *ρ-fractured* if every model is ρ-dominated on at least one instance. In the presence of fracturing, care must be taken in defining the set of winning models over a race sequence. We do so as follows. We first find a minimum hitting set of winning models $\{a_1, a_2, a_3, ...\}$ which covers all instance races. We then define the set A_i as the set of models that won *every* race that a_i won. The set of sets $\{A_1, A_2, A_3, ...\}$ then gives a summary of the winning models over all fractured parts of the instance space. Note that each $A_i \cap A_j = \emptyset$ (where $i \neq j$) as otherwise we could find a smaller hitting set. Also note that in an unfractured instance space the unique A_1 is simply the set of models which won all races. However, for fractured spaces the set $\{A_i\}$ is not uniquely defined as it depends on the hitting set found: nevertheless it gives us a representation of one particular fracturing of the instance space.

[4] `http://savilerow.cs.st-andrews.ac.uk`

3 Methods for Generating Discriminating Instances

The instance space is defined by the parameters of a problem class. Consider the ESSENCE specifications in Figure 1. Langford's Problem has two independent integer parameters and hence a two-dimensional instance space. The Knapsack Problem is an example of a more complex instance space, consisting of two integers and two functions. The first integer, n, governs the number of items and also the domain of the two given functions, which define the weights and values of those items. Our three methods for generating discriminating instances in these spaces are described below. All run a sequence of races and combine the results following the method described in Section 2.

`Undirected`: For each race in a sequence, `undirected` simply draws a sample from the instance space and runs a race. Section 4 describes our sampling method.

`Markov`: This method is loosely based on the Markov chain Monte Carlo methods used, e.g., to estimate the value of a multi-dimensional integral. We assume that discriminating instances are likely to be found near (by some proximity measure) other discriminating instances, and non-discriminating instances near other non-discriminating instances. This naturally leads to a Markov chain that walks the instance space, is attracted towards known discriminating instances, and is repelled from known non-discriminating instances. Our measure of proximity per parameter type:

Integer The distance is simply the absolute difference between the two values.

Total Function Given functions f and g we compute the distance between $f(i)$ and $g(i)$ for each i where both functions are defined, and aggregate using Euclidean distance. When f and g have different domains of definition, some mappings in f and/or g will be discarded. Suppose we had `given weights : function (total) int(1..n) --> int(1..100)` (as in the Knapsack Problem) with `n` a parameter. If the domain of definition `int(1..n)` differs between f and g, then `n` must differ and this will count towards the instance distance.

Set Given sets S and T, the distance is $\sqrt{|S \setminus T| + |T \setminus S|}$, also Euclidean.

Relation Treating a relation as a set of tuples, we use the distance measure for sets.

To obtain the instance distance, we combine the distance measure for each parameter again using the Euclidean distance. This combines elegantly with the Euclidean distances computed per parameter. An initial instance is sampled using the method described in Section 4. A race is run using this instance and a record taken of its discriminatory quality. Each subsequent instance (sampled using the same method) is accepted or rejected according to the scheme below. If an instance is accepted a race is run with that instance, otherwise another instance is generated, and so on until the required race sequence is complete. We use the following acceptance function, where x_{i-1} is the previous accepted instance and x'_i is the proposed instance.

$$A(x_{i-1}, x'_i) = \frac{G'(x'_i)}{G(x_{i-1})}$$

G' estimates the discriminatory quality of the instance in the interval $[0, 1]$ using the quality values of previously accepted instances found by racing. We define a radius of influence r, which is 10% of the greatest possible distance between any two instances,

LANGFORD'S PROBLEM (CSPLIB 24)

```
given k, n : int(1..)
letting seqLength be k * n
letting seqIndex be domain int(1..seqLength)
find seq : function (total, surjective) seqIndex --> int(1..n)
such that forAll i,j : seqIndex , i < j .
        seq(i) = seq(j) -> seq(i) = j - i - 1
```

THE KNAPSACK PROBLEM

```
given n, totalWeight : int(1..)
given weights, values : function (total) int(1..n) --> int(1..)
find picked: set(maxSize n, minSize 1) of int(1..n)
maximising (sum i in picked . values(i) )
such that (sum i in picked . weights(i) ) <= totalWeight
```

THE PROGRESSIVE PARTY PROBLEM (CSPLIB 14)

```
given n_upper, n_boats, n_periods : int(1..)
letting Boat be domain int(1..n_boats)
given capacity, crew : function (total) Boat --> int(1..n_upper)
where forAll i : Boat . crew(i) <= capacity(i),
find hosts : set of Boat,
     sched : set (size n_periods) of function (total) Boat --> Boat
minimising |hosts|
such that  forAll p in sched . range(p) subsetEq hosts,
   forAll p in sched . forAll h in hosts . p(h) = h,
   forAll p in sched . forAll h in hosts .
     (sum b in preImage(p,h) . crew(b))<= capacity(h),
   forAll b1,b2 : Boat , b1 != b2 .
     (sum p in sched . (p(b1) = p(b2))) <= 1
```

THE WAREHOUSE LOCATION PROBLEM. (CSPLIB 34)

```
given n_upper, n_stores, n_warehouses : int(1..30)
letting Store be domain int(1..n_stores),
        WHouse be domain int(1..n_warehouses)
given capacity : function (total) WHouse --> int(1..n_upper),
      opencost : function (total) WHouse --> int(1..n_upper),
      cost : function (total) tuple (Store, WHouse) --> int(1..n_upper)
find open : function (total) Store --> WHouse
minimising (sum r in range(open). opencost(r)) + sum s : Store . cost((s,open(s)))
such that forAll w : WHouse . |preImage(open,w)| <= capacity(w)
```

Figure 1. Four sample ESSENCE specifications.

using the distance measure above. $G'(x'_i)$ finds the set of all previous accepted instances within distance r of x'_i. If this set is non-empty, $G'(x'_i)$ returns the mean of the true quality values for the set. Otherwise, $G'(x'_i) = 0.5$. $G(x_{i-1})$ gives the true quality of x_{i-1}. Finally, a pseudorandom number a is generated within $[0, 1]$, and x'_i is accepted if $A(x_{i-1}, x'_i) \geq a$ then. Hence, the proposed instance is always accepted if $G'(x'_i)$ is greater than $G(x_{i-1})$.

`Smac`**:** Our final method is based on SMAC [15], an automatic algorithm configuration system. Given an algorithm, a description of its parameters, and a set of instances, it finds the set of parameters for the algorithm that delivers the best performance on the set of instances. Finding discriminatory instances is a very similar setting – the algorithm is the problem class specification, its parameters instantiate particular instances of the class and the set of problem "instances" is the set of models. We want to find the set of problem class parameters – the set of problem instances – that has the optimum discriminatory power with respect to the models.

We encode the problem class parameters into SMAC's input format, which uses integer and categorical variables. Integer `given`s are converted to integer parameters with the range specified in the problem definition. We model structured `given`s such as functions using multiple SMAC parameters. When one `given` depends on another, such as the size of the domain of the function depending on n in the Knapsack Problem, we must be conservative: sufficient SMAC parameters are used to accommodate the maximum size of the structured `given`. The extraneous parameters are ignored when racing the instances so produced. Although SMAC has demonstrated that it is able to handle large parameter spaces, this conservative encoding may hinder its ability to cover the space effectively, since many of the values it is producing may be ignored. Furthermore, SMAC does not support the complex constraints between parameters that we sometimes require (`where`), so we use CONJURE to validate the instances that SMAC generates and discard those that do not satisfy the constraints.

4 Uniform Versus Non-uniform Sampling

Except for SMAC, which has its own sampling method, our generation approaches require the ability to sample from the instance space associated with a problem class. Since the instance spaces of the problem classes we consider are typically infinite, our method requires some sensible bounds on the parameters involved in order to circumscribe a finite sub-space of interest. For example in Langford's Problem, as discussed in Section 5, we limit the two integer parameters to the ranges 2..10 and 1..50 respectively.

When the parameters defining the space are independent (e.g. the pair of integer parameters to Langford's problem), uniform sampling is straightforward: simply generate a value uniformly and independently for each parameter. When the parameters are not independent, uniform sampling is more difficult. Reconsider the Knapsack Problem from Figure 1. An approach to sampling from this space is first to generate n uniformly then uniformly and independently generate a mapping for each element of the domain ($1..n$) of the two functions `weights` and `values`. However, this introduces bias: there are many more possible functions for large values of n than there are for small values of n. Hence, if we generate n uniformly then a particular function for small n is far more likely to be selected than a particular function for large n.

A solution to this problem is to enumerate all of the valid instances of the instance sub-space and sample uniformly from this set. This enumeration problem is naturally cast as a (simple) constraint problem. An ESSENCE specification E^* for the enumeration problem can be obtained automatically from an original ESSENCE specification E simply by replacing in E the `given` (parameter) statements with `find` (decision variable) statements and discarding the rest of E. As a very simple example, performing this process for Langford's Problem produces:

```
find k : int(2..10)
find n : int(1..50)
```

Care must be taken that this transformation produces valid ESSENCE. For the Knapsack Problem (assuming sensible parameter limits) it produces:

```
find n : int(1..100)
```

```
find totalWeight : int (1..1000)
find weights : function (total) int(1..n) --> int(1..100)
find values : function (total) int(1..n) --> int(1..100)
```

This is invalid because a decision variable is used to define the size of the domain of the two function variables. The solution we adopt is to leave n as a parameter, solve the problem for each value of n and take the union of the results.

We obtain a single model for E^* automatically using CONJURE and the Compact heuristic [1] — this is sufficient because the enumeration problem is typically easy. MINION is then used to find all solutions to the model. Our `uniform` sampling method is to select uniformly from this set of solutions.

The drawback of `uniform` is that it limits the size of space that we can consider. An alternative approach, which we call `solver-random`, is to sample by requesting a single solution from MINION, employing a random variable and value order. This introduces bias for much the same reason as described above: the distribution of solutions to the model may not be uniform. However, `solver-random` is much more scalable.

In order to compare `uniform` with `solver-random` we performed an experiment on two problem classes: Warehouse Location and the Progressive Party Problem (see Figure 1). So that `uniform` sampling was feasible, the instance space for each problem class was restricted to around a billion instances by limiting the upper bounds of each of their parameters. For each problem class we ran three sequences of thirty races using our `Markov` and `Undirected` generation approaches. In this experiment there was no substantive difference between `Markov` and `Undirected` — probably because of the (lack of) discriminatory quality of the instance sampled as we discuss below. For Warehouse Location the performance of the two sampling methods is identical, reducing 128 input models to 64 non-dominated models. For the Progressive Party Problem `solver-random` actually performs better than `uniform`, reducing 256 input models to 16 non-dominated models, whereas `uniform` produces between 63 and 77 non-dominated models. The bias inherent to `solver-uniform` appears to have been beneficial in this case, guiding the methods to discriminating instances.

These experiments provide some evidence that `solver-random` is a reasonable approach to sampling the instance space. It is also worth noting that the restrictions on the size of the instance space that we explore (to accommodate `uniform`) restricts the discriminatory quality of the instances that we find — producing 64 non-dominated models for Warehouse Location is a relatively weak result. `solver-random` scales easily to more challenging, hence more discriminating, instances as we will see in the following section where we use `solver-random` exclusively.

5 Experimental Results

In this section we report an experimental evaluation of our instance generation methods. Following the outcome of our experiment in Section 4, we use `solver-random` sampling throughout. We experiment on six problem classes described below with results summarised in Table 1. For each class we run three independent sequences of races. For `Markov` and `Undirected` we run 30 races, each with a time budget of 6 hours. This time budget is divided by the number of models to obtain the maximum time allowed for a particular model to solve an instance. Rather than a maximum number of

runs, SMAC requires a total time budget. We specify the maximum time either of the other methods took to complete the race sequence.

Given the volume of experiments we use heterogeneous compute resources, while ensuring that all experiments for each problem class are run on the same resource. In all cases, a model is given 6.5GB of RAM and run on an AMD-based architecture. The Social Golfer Problem was run on a 32-core 2.1GHz machine, Warehouse Location on a 64-core 2.1GHz machine, and the remainder on 2.1GHz processors on Microsoft Azure.

Our results are summarised in Table 1. For each problem class we record: the number of models refined by CONJURE from the associated ESSENCE specification; the size of the output set(s) of models as described in Section 2; the number of discovered fractured parts of the instance space; and, for `Markov` and `Undirected`, the number of races until convergence. This last measure indicates how many of the 30 races are necessary to achieve the final result. Detailed discussion follows.

The Knapsack Problem The parameter space is limited as described in Section 4. All of our methods perform well on this problem class, identifying that the instance space is fractured into two parts. The parts correspond to satisfiable versus unsatisfiable instances. For the satisfiable part, all three methods returned a single winner model (from the 64 input), which employs a 0/1 model of the problem. For the unsatisfiable part, all three methods also returned a single winning model based on an explicit representation of the set of items in the knapsack. Both `Markov` and `Undirected` show some variability in how quickly they converge on this result.

Langford's Problem The parameters to this problem are a pair of independent integers (limited as described in Section 4). Therefore, `solver-random` sampling is unbiased for this problem. None of our methods found the instance space to be fractured, and each was able to reduce the input set of models drastically to a small set of non-dominated models. Over the three independent runs, `Markov` shows a slight edge in performance returning a single model in one case, whereas `Undirected` on one occasion returns six models. Both show some variability in steps to convergence.

The Social Golfer Problem The parameters to this problem are a triple of independent integers, hence `solver-random` sampling is again unbiased. Each parameter is limited between 1 and 100. Hence, a large fraction of this instance space is unfeasibly difficult so we would expect our more informed approaches to perform better in this case study. In fact, SMAC struggles with this class, in two cases finding no discriminating instances at all. `Undirected`'s performance is variable, sometimes outputting a single model, but on one occasion also finding no discriminating instances. By contrast `Markov` performs well, reducing the 24 input models to between 1 and 4 on each run and on one occasion identifying a fracture in the instance space.

The Progressive Party Problem This class has the most (256) input models. Perhaps unsurprisingly, therefore, the instance space is fractured — but this is only identified by `Markov` and `Undirected`. When fracturing is detected, `Markov` is more consistent than `Undirected` in the size of the returned set for the first fractured part.

The Warehouse Location Problem All three methods are able to reduce the input set of 128 models significantly. `Markov` has the best performance, followed by SMAC and then `Undirected`. No fracturing was found.

		Markov			SMAC		Undirected		
Problem	#Models	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Knapsack-1	64	1,1	2	20	1,1	2	1,1	2	7
Knapsack-2	64	1,1	2	13	1,1	2	1,1	2	30
Knapsack-3	64	1,1	2	2	1,1	2	1,1	2	3
Langford-1	154	4	1	28	4	1	6	1	1
Langford-2	154	1	1	14	3	1	4	1	14
Langford-3	154	4	1	2	4	1	3	1	29
SGP-1	24	1,4	2	26	1	1	1	1	7
SGP-2	24	4	1	13	24	1	1	1	15
SGP-3	24	1	1	7	24	1	24	1	30
PPP-1	256	32,8	2	2	8	1	53,8	2	19
PPP-2	256	32,8	2	12	16	1	27,8	2	11
PPP-3	256	8	1	9	15	1	17	1	7
Warehouse-1	128	32	1	17	32	1	45	1	16
Warehouse-2	128	24	1	13	49	1	72	1	26
Warehouse-3	128	8	1	16	27	1	54	1	26
BACP-1	48	1,1	2	24	26	1	1	1	8
BACP-2	48	1	1	1	8	1	1	1	25
BACP-3	48	1	1	2	25	1	1	1	7

Table 1. Results for the six problem classes over three independent runs.

Balanced Academic Curriculum Problem Only `Markov` is able to detect a fracture in the instance space for this class, and only on one of its runs. Both `Markov` and `Undirected` reduce the input model set drastically from 48 to a single winning model. SMAC performs much less well on this class, perhaps hampered by the necessary encoding compromises (see Section 3) on what is a more complex instance space defined by 7 integers, two functions and a relation.

6 Conclusions

We have developed and investigated three methods for generating discriminating instances for the purpose of automated constraint model selection. Our experimental evaluation shows that all of these methods are capable of reducing a large number of possible models to a much smaller set. The methods are able to detect fracturing if it occurs and successfully determine the best models for each fraction. Overall, our novel `Markov` approach has the best performance on the problem classes in our experiments.

Acknowledgements This research is supported by UK EPSRC EP/K015745/1. Bilal Syed Hussain is supported by an EPSRC scholarship, and a Google Europe Scholarship. Chris Jefferson is a University Research Fellow funded by the Royal Society. Lars Kotthoff is supported by EU FP7 grant 284715. The Microsoft Azure processors were provided by a Microsoft Azure for Research grant.

References

1. Akgun, O., Frisch, A.M., Hussain, B.S., Gent, I.P., Jefferson, C.A., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: 19th International Conference on Principles and Practice of Constraint Programming. pp. 107–116 (2013)
2. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: 25th Conference on Artificial Intelligence (AAAI) (2011)
3. 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)
4. 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. pp. 123–137 (2004)
5. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: Acquiring constraint networks using a SAT-based version space algorithm. In: 21st Conference on Artificial Intelligence (AAAI). pp. 1565–1568 (2006)
6. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: The Genetic and Evolutionary Computation Conference (GECCO). vol. 2, pp. 11–18 (2002)
7. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: 17th European Conference on Artificial Intelligence (ECAI). pp. 73–77 (2006)
8. 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. pp. 812–816 (2003)
9. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Logic Based Program Synthesis and Transformation (LOPSTR). pp. 214–232 (2003)
10. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: 19th International Joint Conference on Artificial Intelligence (IJCAI). pp. 109–116 (2005)
11. 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)
12. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: 17th European Conference on Artificial Intelligence (ECAI). vol. 141, pp. 98–102 (2006)
13. 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)
14. Hnich, B.: Function variables for constraint programming. AI Communications 16(2), 131–132 (2003)
15. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: 5th Conference on Learning and Intelligent Optimization (LION). pp. 507–523 (2011)
16. Koninck, L.D., Brand, S., Stuckey, P.J.: Data independent type reduction for Zinc. In: 9th International Workshop on Constraint Modelling and Reformulation (2010)
17. 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)
18. Little, J., Gebruers, C., Bridge, D.G., Freuder, E.C.: Using case-based reasoning to write constraint programs. In: 9th International Conference on Principles and Practice of Constraint Programming. p. 983 (2003)

19. 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), 229–267 (2008)
20. 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)
21. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack., G.: Minizinc: Towards a standard CP modelling language. In: 13th International Conference on the Principles and Practice of Constraint Programming. pp. 529–543 (2007)
22. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C.A., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: 20th International Conference on Principles and Practice of Constraint Programming (September 2014)
23. Rendl, A.: Effective Compilation of Constraint Models. Ph.D. thesis, University of St Andrews (2010)
24. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA (1999)