A Model Seeker: Extracting Global Constraint Models From Positive Examples

Nicolas Beldiceanu[1] and Helmut Simonis[2]

[1] TASC team (INRIA/CNRS), Mines de Nantes, France
Nicolas.Beldiceanu@mines-nantes.fr
[2] Cork Constraint Computation Centre **
Department of Computer Science, University College Cork, Ireland
h.simonis@4c.ucc.ie

Abstract. We describe a system which generates finite domain constraint models from positive example solutions, for highly structured problems. The system is based on the global constraint catalog, providing the library of constraints that can be used in modeling, and the Constraint Seeker tool, which finds a ranked list of matching constraints given one or more sample call patterns.
We have tested the modeler with 230 examples, ranging from 4 to 6,500 variables, using between 1 and 7,000 samples. These examples come from a variety of domains, including puzzles, sports-scheduling, packing & placement, and design theory. When comparing against manually specified "canonical" models for the examples, we achieve a hit rate of 50%, processing the complete benchmark set in less than one hour on a laptop. Surprisingly, in many cases the system finds usable candidate lists even when working with a single, positive example.

1 Introduction

In this paper we present the *Model Seeker* system which generates constraint models from example solutions. We focus on problems with a regular structure (this encompasses *matrix models* [14]), whose models can be compactly represented as a small set of conjunctions of identical constraints. We exploit this structure in our learning algorithm to focus the search for the strongest (i.e. most restrictive) possible model.

In our system we use global constraints from the global constraint catalog [2] mainly as modeling constructs, and not as a source of filtering algorithms. The global constraints are the primitives from which our models are created, each capturing some particular aspect of the overall problem. Using existing work on global constraints for mixed integer programming [20] or constraint based local search [16], our results are not only applicable for finite domain constraint programming, but can potentially reach a wider audience.

The input format we have chosen consists of a flat vector of integer values, allowing for different representations of the same problem. We do not force the user to adapt his input to any particular technology, but rather aim to be able to handle examples taken from a variety of existing sources.

** The second author is supported by EU FET grant ICON (project number 284715).

In our method we extensively use meta-data about the constraints in the catalog, which describe their properties and their connection. We have added a number of new, useful information classes during our work, which prove to be instrumental in recognizing the structure of different models.

The main contribution of this paper is the presentation of the implemented Model Seeker tool, which can deal with a variety of problem types at a practical scale. The examples we have studied use up to 6,500 variables, and deal with up to 7,000 samples, even though the majority of the problems are restricted to few, and often unique solution samples. We currently only work with positive examples, which seems to provide enough information to achieve quite accurate models of problems. As a side-effect of our work we also have strengthened the constraint description in the constraint catalog with new categories of meta-data, in particular to show implications between different constraints.

Our paper is structured in the following way: We first introduce a running example, that we will use to explain the core features of our system. In Section 2, we describe the basic workflow in our system, also detailing the types of meta-data that are used in its different components. We present an overview of our evaluation in Section 3, which is followed by a discussion of related work (Section 4), before finishing with limitations and possible future work in Section 5. For space reasons we can only give an overview of the learning algorithm and the obtained results. A full description can be found in a companion technical report at `http://4c.ucc.ie/~hsimonis/modelling/report.pdf`.

1.1 A Running Example

As a running example we use the 2010/2011 season schedule of the Bundesliga, the German soccer championship. We take the data given in `http://www.weltfussball.de/alle_spiele/bundesliga-2010-2011/`, replacing team names with numbers from 1 to 18. The schedule is given as a set of games on each day of the season. Table 1 shows days 1, 2, 3, 18 and 19 of the schedule. Each line shows all games of one day; on the first day, team 1 (at home) is playing against team 2 (away), team 3 (at home) plays team 4, etc. The second half of the season (days 18-34) repeats the games of the first half, exchanging the home and away teams, on day 18, for example, team 18 (at home) plays team 17, team 2 (at home) plays team 1, and so on. Overall, each team plays each other twice, once at home, and once away in a double round-robin scheme.

As input data we receive the flat vector of numbers, we will reconstruct the matrix as part of our analysis. Note that for most sports scheduling problems we will have access to only one example solution, the published schedule for a given year, schedules from different years encode different teams and constraints, and are thus incomparable.

2 Workflow

We will now describe how we proceed from the given positive examples to a candidate list of constraints modeling the problem. The workflow is described in Figure 1. Data are shown in green, software modules in blue/bold outline, and specific global constraint

Table 1. Bundesliga Running Example: Input Data

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
8	1	14	11	4	7	2	15	12	13	6	9	10	3	18	5	16	17
3	14	17	2	13	6	5	12	9	16	11	18	1	4	15	8	7	10
							...										
18	17	2	1	4	3	6	5	10	9	16	15	14	13	12	11	8	7
13	12	11	14	17	16	15	2	9	6	1	8	7	4	5	18	3	10
							...										

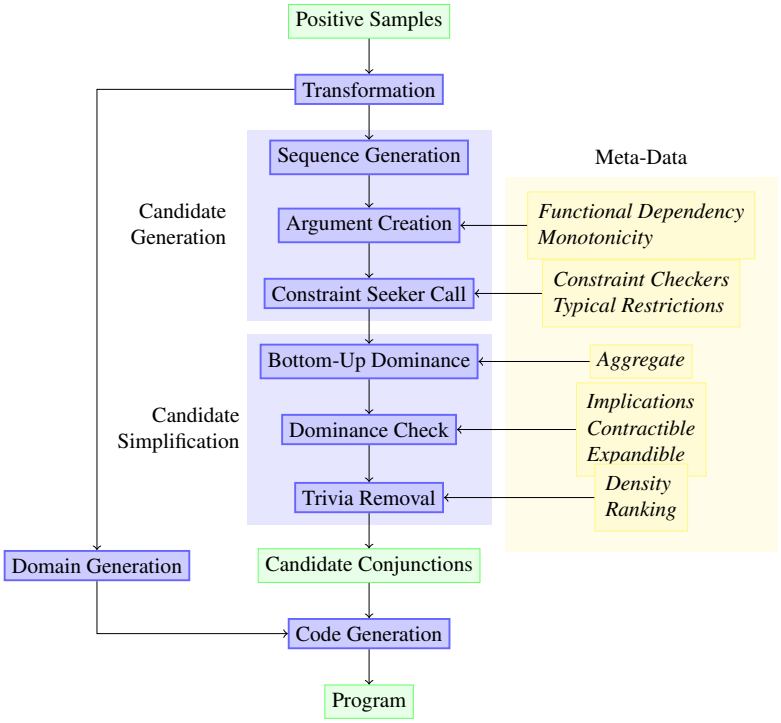
catalog meta-data are shown in yellow/italics. We first give a brief overview of the modules, and then discuss each step in more detail.

Transformation In a first step, we try to convert the input samples to other, more appropriate representations. This might involve replacing a 0/1 format with finite domain values, or converting different graph representations into the successor variable form used by the global constraints in the catalog. For some transformations, we keep both the original and the transformed representation for further analysis, for others we replace the original sample with the transformed data.

Candidate Generation The second step (Sequence Generation) tries to group the variables of the sample into regular subsets, for example interpreting the input vector as a matrix and creating subsequences for all rows and all columns. In the Argument Creation step, we create call patterns for constraints from the subsequences. We can try each subsequence on its own, or combine pairs of subsequences or use all subsequences together in a single collection. We also try to add additional arguments based on functional dependencies and monotonic arguments of constraints, described as meta-data in the global constraint catalog. For each of these generated call patterns, we then call the Constraint Seeker to find matching constraints, which satisfy all subsequences of all samples. For this we enforce the *Typical Restrictions*, meta-data in the catalog, which describe how a constraint is typically used in a constraint program. Only the highest ranking candidates are retained for further analysis.

Candidate Simplification After the seeker calls, we potentially have a very large list of possible candidate conjunctions (up to 2,000 conjunctions in our examples), we now have to reduce this set as much as possible. We first apply a Dominance Check to remove all conjunctions of constraints that we can show are implied by other conjunctions of constraints in our candidate list. Instead of showing the implication from first principles, we rely on additional meta-data in the catalog, which describe implications between constraints, but we also use conditional implications which only hold if certain argument restrictions apply, and expandible and contractible [22] properties, which state that a constraint still holds if we add or remove some of its decision variables. The dominance check is the core of our modeling system, helping to remove irrelevant constraints from the candidate list. In the last step before our final candidate list output the system removes trivial constraints and simplifies some constraint pattern. This also performs a ranking of the candidates based on the constraint and sequence generator used,

Fig. 1. Workflow in the Model Seeker



trying to place the most relevant conjunction of constraints at the top of the candidate list.

Code Generation As a side effect of the initial transformation, we also create potential domains for the variables of our problem. In the default case, we just use the range of all values occurring in the samples, but for some graph-based transformations a more refined domain definition is used. Given the candidate list and domains for the variables, we can easily generate code for a model using the constraints. At the moment, we produce SICStus Prolog code using the call format of the catalog. The generated code can then be used to validate the samples given, or to find solutions for the model that has been found.

After this brief overview, we will now discuss the different process steps in more detail.

2.1 Transformation

In finite domain programming, there are implicit conventions on how to express models leading to effective solution generation. In our system, we can not assume that the user is aware of these conventions, nor that the sample solutions are already provided

in the "correct" form. We therefore try to apply a series of (currently 12) input transformations, that convert between different possible representations of problems, and that retain the form which matches the format expected by the constraint catalog. In each case, some pre- and post-conditions must be satisfied for the transformation to be considered valid. We now give some examples.

Converting 0/1 Samples If a solution is given using only 0/1 values, there may be a way of re-interpreting the sample as a permutation with finite domain variables. If we consider the 0/1 values as an $n \times n$ matrix (x_{ij}) where each row and each column contains a single one, we can transform this into a vector v_i of n finite domain values based on the equivalence

$$\forall_{1 \leq i \leq n} \forall_{1 \leq j \leq n} : \quad x_{ij} = 1 \Rightarrow v_i = j$$

This transformation is the equivalent of a channeling constraint between 0/1 and finite domain variables, described for example in [19].

Using successor notation for graph constraints Most graph constraints in the catalog use a *successor* notation to describe circuits, cycles or paths, i.e. the value of a node indicates the next node in the final assignment. But this is not the only way of describing graphs. In the original Knight's Tour formulation [31], the value of a node is the position in the path, i.e. the first node has value one, the second value two, and so on. We have defined transformations which map between these (and several other) formats, while checking that the resulting graph can be compactly described.

Table 2. Bundesliga Running Example: Transformed Problem

2	−1	4	−3	6	−5	8	−7	10	−9	12	−11	14	−13	16	−15	18	−17
−8	15	−10	7	−18	9	−4	1	−6	3	−14	13	−12	11	−2	17	−16	5
4	−17	14	−1	12	−13	10	−15	16	−7	18	−5	6	−3	8	−9	2	−11
								...									
−2	1	−4	3	−6	5	−8	7	−10	9	−12	11	−14	13	−16	15	−18	17
8	−15	10	−7	18	−9	4	−1	6	−3	14	−13	12	−11	2	−17	16	−5
								...									

Using Schreuder tables Another transformation is linked to sports scheduling problems. In many cases, users like to give the schedule as a list of fixtures, listing which games will be played on each day. The first team is the home team, the second the away team. For constraint models, the format of Schreuder tables [28], as shown in Table 2 for our running example, can lead to more compact models [24, 30, 17, 18]. For each time-point t over q rounds and for an even number of teams n, they can be obtained from the fixture representation as follows:

$$\forall_{1 \leq t \leq q(n-1)} \forall_{1 \leq i \leq \lfloor n/2 \rfloor} : \quad x_{2i-1,t} = k, x_{2i,t} = l \Rightarrow v_{k,t} = l, v_{l,t} = -k$$

2.2 Sequence Generator

After the input transformation, we have to consider possible, regular substructures which group the samples into subsequences. For space reasons again, we only give some examples of the sequence generators used in our running example, the full list (containing 21 generators) with their formal definition can be found in the technical report, some were already described in [4].

vector(n) This is the most basic sequence generator of treating all elements of the sample as a single sequence of size n.

scheme(n, r, c, a, b) By far the most common sequence generator treats a sample of length n as an $r \times c$ matrix, and creates non-overlapping blocks of size $a \times b$, creating n/ab sequences of size ab. The number of such partitions depends on the number of factors of n, as $n = rc$. For our running example (Section 1.1) with 612 values, we have to consider the matrices 2×306, 3×204, 4×153, 6×102, 9×68, 17×36, 18×34, 34×18, 36×17, 68×9, 102×6, 153×4, 204×3 and 306×2. Some of the blocks created from these matrices lead to the same partition of the variables, only one representative is kept.

repart(n, r, c, a, b) This sequence generator also treats the sample of size n as a $r \times c$ matrix, and considers blocks of size $a \times b$. But it groups elements in the same position from each block, creating $a \times b$ sequences of size $n/(ab)$.

For the running example, a total of 68 subsequence collections are generated. Note that the subsequences often, but not always, have the same size. We also provide an API where the user can provide his own sequence generators, this can be helpful to deal with known, but irregular structure in the problem.

2.3 Argument Creation

In the next step of the operation, we convert the generated subsequences into call patterns for the Constraint Seeker [3]. In order to consider more of the constraints in the catalog, we have to provide different argument signatures by organizing the subsequences in different ways, and by adding arguments.

Single, Pairs and Collection In the first part we decide how we want to use the subsequences. Consider we have k subsequences, each of length m. If we use each subsequence on its own, we create k call patterns with a single argument, each a collection of m variables. This corresponds to the argument pattern used by `alldifferent` constraints, for example. We can also consider pairs of subsequences, creating a call patterns with two arguments, for $k-1$ calls to a predicate like `lex_greater`. Finally we can use all subsequences as a single collection of collections, which creates one call with a collection of k collections of m elements each. This would match a constraint like `lex_alldifferent`. We generate all these potential calls in parallel, and perform the steps described in the following two paragraphs.

Value Projection For some problems (like our transformed, running example), a projection from the original domain to a smaller domain can lead to a more compact model. If, for example, some of the values in the sample are positive, and others are negative, we can try a projection using the absolute value or the sign of the numbers, in addition to the original values.

Adding Arguments Many global constraints use additional arguments besides the main decision variables. If we do not generate these arguments in the call pattern, we can not find such constraints with the Constraint Seeker. But just enumerating all possible values for these additional arguments would lead to a combinatorial explosion. Fortunately, we can compute values for these arguments in case of functional dependencies and monotonic arguments. This is similar to the argument generation discussed for the `gcc` constraint discussed in [8].

2.4 Constraint Seeker

The Constraint Seeker [3] will find a ranked list of global constraints that satisfy a collection of positive and negative sample calls, using the available constraint checkers of the catalog. We use this seeker as a black-box for all call patterns with all additional argument values and value projections defined in the previous section.

Using Multiple, Positive Samples The seeker first checks that the call signature matches the constraint, then tries to evaluate the constraint on the samples. In our case, these are the call patterns prepared in the previous step for all subsequences of all positive examples given. In our modeling system we currently do not consider negative examples. They would require a slightly different treatment, as a negative example can be rejected by just one constraint, while all positive examples must be accepted by all constraints found.

Typical Restrictions In addition to the restrictions that must hold for the constraint to be applied, in our modeling tool we also check for the *typical* restrictions that are specified in the catalog. The `alldifferent` constraint for example can be called with an empty collection, but a typical use would have more than two variables in the collection. The typical constraints are expressed using the same language as the formal restrictions of the catalog, checking their validity thus does not require any additional code.

Selecting Top-Ranked Elements The Constraint Seeker returns a ranked list of candidates, this ranking is a combination of structural properties (functional dependencies or monotonic arguments), implications between constraints, estimated solution density and estimated popularity of the constraint described in [3]. In our system we only use the top ranked element that satisfies all subsequences of all samples. This reduces the number of candidates to be considered, while at the same time it does not seem to exclude constraints that are required for the highly structured problems considered.

For our running example, we perform 1,099 calls to the Constraint Seeker, which performs 82,458 constraint checks, and which results in 589 possible candidate conjunctions. We now face the task of reducing this candidate list as much as possible, keeping only interesting conjunctions.

2.5 Bottom-up Dominance

Some constraints like `sum` or `gcc` have the *aggregate* property, one can combine multiple such constraints over disjoint variable sets by adding the right hand sides or summing the counter values. As an example, we can combine

$$x_1 + x_2 = 5 \wedge x_3 + x_4 = 2 \Rightarrow x_1 + x_2 + x_3 + x_4 = 7$$

We want to remove aggregated constraints of this type, as they are implied by conjunctions of smaller constraints. We perform a bottom-up saturation of combining constraints with the aggregate property up to a limited size, and remove any candidate conjunctions where all constraints are dominated.

2.6 Dominance Check

The dominance check compares all conjunction candidates against each other (worst case quadratic number of comparisons), and marks dominated entries. Note that dominated entries may be used to dominate other entries, and thus can not be removed immediately. We use a number of meta-data fields to check for dominance.

Implications In our final candidate list, we are interested in only the strongest, most restrictive constraints, all constraints that are implied by other candidate constraints can be excluded. Note that this will sometimes lead to overly restrictive solutions, especially if only a few samples are given.

Checking if some conjunction is implied by some other conjunction for a particular set of input values is a complex problem, a general solution would require sophisticated theorem proving tools like those used in [11] for a restricted problem domain. We do not attempt such a generic solution, but instead rely on meta-data in the catalog linking the constraints. That meta-data is useful also for understanding the relations between constraints, and thus serves multiple purposes. This syntactic implication check is easy to implement, but only can be used if both constraints have the same arguments.

Conditional Implications For some constraints additional implications exist, but only if certain restrictions apply. The `cycle` constraint for example implies the `circuit` constraint, but only if the `NCYCLE` argument is equal to one. For conditional implications the arguments do not have to be the same, but the main decision variables used must match.

Contractibility and Expandability Other useful properties are contractibility [22] and expandibility. A constraint (like `alldifferent`) is *contractible* if it still holds if we remove some of its decision variables. This allows us to dominate large conjunctions of constraints with few variables with small conjunctions of constraints with many variables. Due to the way we systematically generate all subsequence collections, this is often possible. In a similar way, some constraints like `atleast` are *expandible*, they still hold if we add decision variables. We can again use this property to dominate some conjunctions of constraints. Details and possible extensions have been described in [4].

Hand-coded Domination Rules Some dominance rules are currently hand-crafted in the program, if the required meta-data have not yet been formalized in the catalog description. Such examples can be an important source of requirements for the catalog itself, enhancing the expressive power of the constraint descriptions.

2.7 Trivia Removal

Even after the dominance check, we can still have candidate explanations which are valid and not dominated, but which are not useful for modeling. In the trivia removal section, we eliminate or replace most of these based on sets of rules.

Functional dependencies on single samples In Section 2.3 we have described how we can add some arguments to a call pattern for functional dependencies. In the case of pure functional dependencies [1], we have to worry about pattern consisting of a single subsequence with a single sample. In that case, the constraint does not filter any pattern, as for each pattern the correct value can be selected. We therefore remove such candidates.

Constraint Simplification At this point we can also try to simplify some constraints that have particular structure. A typical example are `lex_chain` constraints on a subsequence, where already the first entries of the collections are ordered in strictly increasing order. We can therefore replace the `lex_chain` constraint on the subsequences with a `strictly_increasing` constraint on the first elements of the collections, using a special `first` sequence generator. These constraints often occur as symmetry-breaking restrictions in models, which we find if all the samples given respect the symmetry breaking.

Uninteresting Constraints Even with the typical restrictions in the Constraint Seeker, we often find candidates (like `not_all_equal`) which are not very interesting for defining models. As a final safe-guard, we use a black-list to remove some combinations of constraints and sequence generators that should not be included in our models.

2.8 Candidate List for Bundesliga Schedule

Table 3 shows the list of the candidate conjunctions generated for our transformed example problem. Entries in green match a manually defined model, ten other candidates are also proposed. The arguments of constraints in the Constraint Conjunction column indicate any additional parameters, the $*n$ indicates how many constraints form the conjunction. The value projections `absolute_value` and `sign` convert each element of the input data, `id` denotes the identity projection.

Some of the constraints mentioned are perhaps unfamiliar, we provide a short definition. The constraint `symmetric_alldifferent`$([x_1, .., x_n])$ [2, page 1854] in line 4 states that

$$\forall_{1 \leq i \leq n} : \quad x_i \in [1, n]; x_i = j \iff x_j = i$$

It expresses the constraint that if team A plays team B on some day, then team B will play team A. The constraints `twin`$([\langle x_1, y_1 \rangle, ..., \langle x_n, y_n \rangle])$ [2, page 1896] in lines 7, 19 and 20 state that

$$\forall_{1 \leq i \leq n} : \quad (x_i = u \wedge y_i = v) \Rightarrow (\forall_{1 \leq j \leq n} : \; x_j = u \iff y_j = v)$$

These constraints express the fact that the tournament is played in two symmetric half-seasons, with home and away games swapped. Note that constraints 8, 21 and 23 also express this condition, but using an `elements` constraint, pairing positive and negative numbers. The `alldifferent` constraint in line 1 expresses that no repeat games occur in the season, while that of line 5 states that all teams play on each day. The `strictly_increasing` constraint in line 9 results from the simplification of a symmetry breaking `lex_chain` constraint. The `gcc` in line 14 states that each team plays 17 home (positive value) and 17 away (negative value) games. Finally, the `among_seq` constraint in line 22 states that no team has more than two consecutive away games.

Table 3. Constraint Conjunctions for Problem Bundesliga

-	Sequence Generator	Projection	Constraint Conjunction
1	scheme(612,34,18,34,1)	id	alldifferent*18
2	scheme(612,34,18,2,2)	id	alldifferent*153
3	scheme(612,34,18,1,18)	id	alldifferent*34
4	scheme(612,34,18,1,18)	absolute_value	symmetric_alldifferent([1..18])*34
5	scheme(612,34,18,17,1)	absolute_value	alldifferent*36
6	repart(612,34,18,34,9)	id	sum_ctr(0)*306
7	repart(612,34,18,34,9)	id	twin*1
8	repart(612,34,18,34,9)	id	elements([*i*,-*i*])*1
9	first(9,[1,3,5,7,9,11,13,15,17])	id	strictly_increasing*1
10	vector(612)	id	global_cardinality([-18.. -1-17,0-0,1..18-17])*1
11	repart(612,34,18,34,9)	id	sum_powers5_ctr(0)*306
12	repart(612,34,18,34,9)	id	sum_cubes_ctr(0)*306
13	repart(612,34,18,34,3)	sign	global_cardinality([-1-3,0-0,1-3])*102
14	scheme(612,34,18,34,1)	sign	global_cardinality([-1-17,0-0,1-17])*18
15	repart(612,34,18,17,9)	sign	global_cardinality([-1-2,0-0,1-2])*153
16	repart(612,34,18,2,9)	sign	global_cardinality([-1-17,0-0,1-17])*18
17	scheme(612,34,18,1,18)	sign	global_cardinality([-1-9,0-0,1-9])*34
18	repart(612,34,18,34,9)	sign	sum_ctr(0)*306
19	repart(612,34,18,34,9)	sign	twin*1
20	repart(612,34,18,34,9)	absolute_value	twin*1
21	repart(612,34,18,34,9)	sign	elements([*i*,-*i*])*1
22	scheme(612,34,18,34,1)	sign	among_seq(3,[-1])*18
23	repart(612,34,18,34,9)	absolute_value	elements([*i*,*i*])*1
24	first(9,[1,3,5,7,9,11,13,15,17])	absolute_value	strictly_increasing*1
25	first(6,[1,4,7,10,13,16])	absolute_value	strictly_increasing*1
26	scheme(612,34,18,34,1)	absolute_value	nvalue(17)*18

2.9 Domain Creation

By default, the domains of the variables in our generated models are the interval between the smallest and largest value occurring in the samples. Based on the transformation used, we can use more restricted domains for graph models like graph partitioning and domination [15], where the domain of each variable/node specifies the initial graph.

2.10 Code Generation

The code generation builds flat models for the given instances. The programs consist of five parts, we first define all variables with their domains, then state all restrictions due to fixed values as assignments, state any projections used to simplify the variables, then build the constraints in the catalog syntax, and finally call a generic value assignment routine to search for a solution. We can use the generated model as a test to check if it accepts the given samples, or to generate new solutions for the problem. Many puzzles have a unique solution, we can count solutions of our program to see if the generated model is restrictive enough to capture this property.

It would be straightforward to generate the code for other systems than SICStus Prolog, provided that the catalog constraints are supported. A version generating FlatZ-

inc[23] or XCSP [27] would be especially attractive to benefit from the variety of back-end solvers which support these formats.

3 Evaluation

Table 4 shows summary results for selected problems of our evaluation set. The problems range from sports scheduling (ACC Basketball Scheduling, csplib11; Bundesliga; DEL2011 (German ice hockey league); Scotish Premier League (soccer); Rabodirect Pro 12 (rugby)), to scheduling (Job-shop 10x10 [10]) and placement (Duijvestijn, csplib9; Conway 5x5x5 [5]; Costas Array [12]), design theory (BIBD, csplib28; Kirkman [13]; Orthogonal Latin Squares [9]), event scheduling (Social Golfer, csplib10; Progressive Party, csplib13) and puzzles. Details of these problems can be found in the technical report mentioned before. Smaller problems are solved within seconds, even the largest require less than 5 minutes on a single core of a MacBook Pro (2.2GHz) with 8Gb of memory.

The columns denote: *Transformation Id*: the number of the transformation applied (if any), *Instance Size*: the number of values in the solution, i.e. the number of variables in the model, *Nr Samples*: the number of solutions given as input, *Nr Sequences*: the number of sequence sets generated, *Nr Seeker Calls*: the number of times the Constraint Seeker is called, *Constraint Checks*: the number of calls to constraint checkers inside the seeker, *Nr Relevant*: the number of initial candidate conjunctions found by the Constraint Seeker, *Nr Non Dom*: the number of non-dominated candidates remaining after the dominance checkers, *Nr Specified*: the number of conjunctions specified in the manual, "canonical" model, *Nr Models*: the number of conjunctions given as output of the Model Seeker, *Nr Missing*: how many of the manually defined conjunctions were not found by the system, *Hit Rate*: the percentage rate of Nr Specified to Nr Models, a value of 100% indicates that exactly the candidates of the canonical model were found, and *Time*: the execution time in seconds.

For two of the problems, we only find part of the complete model. The Progressive party problem [29] requires a bin-packing constraint that we currently do not recognize, as it relies on additional data for the boat sizes, while the ACC basketball problem contains several constraints which apply only for specific parts of the schedule, and which can not be learned from a single solution. Also note that for the De Jaenisch problem [26], we show results with and without a transformation. This problem combines a "near" magic square, found without transformation, with an Euler Knight tour, using transformation 7.

For our full evaluation, we have used 230 examples from various sources. For 10 of the examples no reasonable model was generated, either because we did not have the right sequence generator, or we are currently missing the global constraint required to express the problem. For a further 37 problems, only part of the model was found. This is typically caused by some constraint requiring additional data, not currently given as input, or by an over-specialization of the output, where the Model Seeker finds a more restrictive constraint than the one specified manually. Overall, we considered 73 constraints in the Constraint Seeker, and selected 53 different global constraints as potential solutions. This is only a fraction of the 380 constraint in the catalog, many of the miss-

Table 4. Selected Example Results

Name	Transformation Id	Instance Size	Nr Samples	Nr Sequences	Nr Seeker Calls	Constraint Checks	Nr Relevant	Nr Non Dom	NrSpecified	Nr Models	Nr Missing	Hit Rate	Time [s]
12 Amazons All	-	12	156	36	127	35596	55	9	5	8	0	62.50	2,64
8 Queens All	-	8	92	20	100	12077	34	7	3	5	0	60.00	0.83
ACC Basketball Schedule	-	162	1	109	1786	29117	772	263	23	36	7	n/a	9.17
BIBD (8,14,7,4,3)	-	112	92	151	626	92461	232	112	4	15	0	26.67	26.01
Bundesliga	18	612	1	68	1099	52933	589	169	16	26	0	61.54	51.44
Conway 5x5x5 Packing	-	102	1	60	184	2619	78	35	1	2	0	50.00	0.42
Costas Array 12	-	12	48	36	121	14820	42	2	2	2	0	100.00	1.01
DEL 2011	-	728	1	235	1334	66999	555	173	3	8	0	37.50	54.23
De Jaenisch Tour	-	64	1	83	568	10130	283	58	2	13	0	15.38	1.66
De Jaenisch Tour	7	64	1	36	219	12952	113	67	1	1	0	100.00	0.46
Dominating Knights 8	9	64	2	36	141	11021	51	42	1	1	0	100.00	0.31
Duijvestijn 21	-	84	1	111	504	11625	240	102	1	12	0	8.33	1.77
Euler Knight Cube 4x4x4	7	64	1	36	208	12759	97	58	1	1	0	100.00	0.42
Job Shop 10x10 (10 sol)	-	400	10	326	1521	80589	582	130	2	2	0	100.00	40.27
Kirkman Wikipedia	-	105	1	40	179	2634	89	39	3	5	0	60.00	1.21
Leaper Tour 18x18	7	324	1	60	298	105955	140	61	1	1	0	100.00	2.18
Magic Square All	-	16	7040	33	176	1068574	57	5	4	4	0	100.00	115.07
Magic Square Duerer	-	16	1	33	212	2074	115	44	9	15	0	60.00	0.25
Orthogonal Latin Squares 10	-	200	1	147	910	15441	443	118	3	8	0	37.50	6.04
Progressive Party	-	174	1	45	171	4279	61	31	4	3	1	n/a	0.70
Rabodirect Pro12	18	264	1	66	1041	46898	539	155	8	13	0	61.54	7.91
Scotish Premier League	18	396	1	68	992	58959	459	157	9	12	0	75.00	14.18
Social Golfer	-	288	1	528	2813	69681	1221	256	5	36	0	13.89	61.93
Sudoku 81x81	-	6561	1	91	657	101075	334	68	3	5	0	60.00	244.16

ing constraints have more complex argument signatures or use finite sets, which are currently not available in SICStus Prolog.

Figure 2 shows the number of candidates found for all examples studied as a function of the instance size, split between single samples and multiple samples. Note that the plot uses a log-log scale. The results indicate that even with a single sample, the number of candidate conjunctions found is quite limited, this drops further if multiple samples are used.

Another view of all the results is shown in Figure 3. It shows the relationship between number of variables and execution time, again grouped by problems with a single sample and problems for which multiple samples were provided. While no formal complexity analysis has been attempted, as several subproblems are expressed as constraint problems, results seem to indicate a low-polynomial link between problem size and execution time. The non-linear least square fit for the single sample problems is $8.5e^{-2}x^{0.90}$, and for multiple samples $6.1e^{-3}x^{1.45}$.

Fig. 2. Candidates as a Function of Problem Size (Variables)

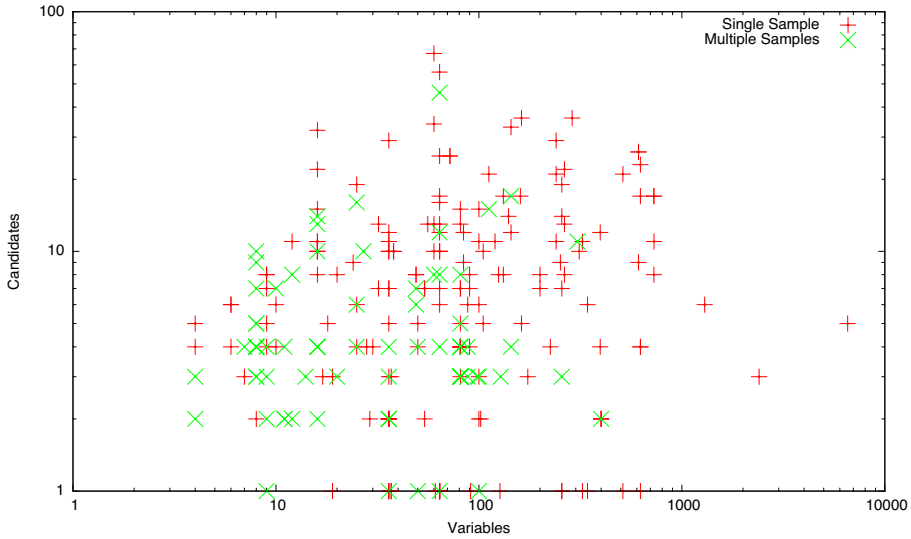


Fig. 3. Execution Time as a Function of Problem Size (Variables)

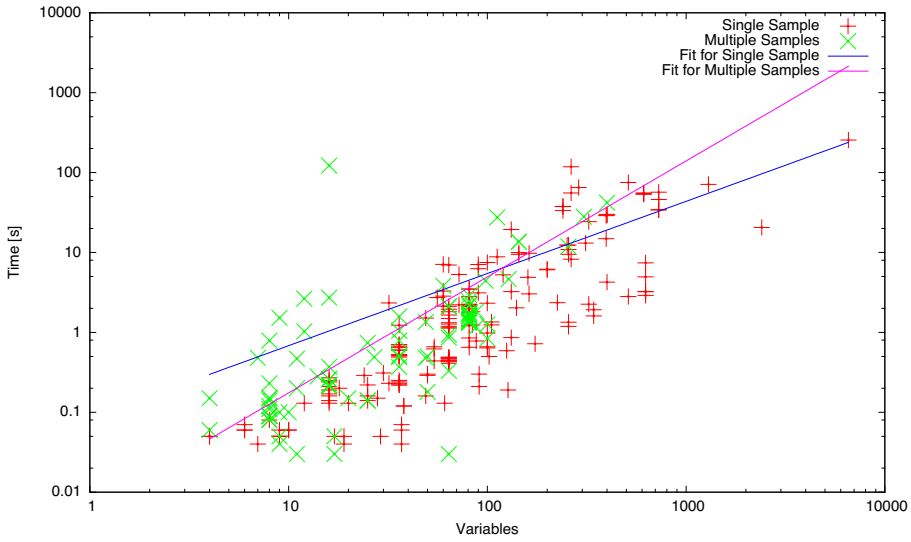


Table 5 shows the number of lines required for the different components of the system, as well as accumulated execution times over all 230 examples measured for these components. The programming effort is fairly evenly split amongst the different components, while the two dominance checkers require nearly two-thirds of the total execution time, with the constraint seeker using another quarter of the time. The system is written in SICStus Prolog 4.2, and uses the Constraint Seeker [3] with an additional 6,500 lines of code, and the global constraint catalog meta-data description of 60,000 lines of Prolog.

Table 5. Lines of Code / Run Time per Module over all 230 Examples

Module	Lines	Time [s]	% of Total
Transformation	1,500	1	0.03
Sequence Generation	1,000	53	2.81
Argument Creation	1,000	150	7.84
Constraint Seeker Call	300	464	24.22
Bottom-Up Check	200	506	26.42
Dominance Check	800	739	38.61
Trivia Removal	500	1	0.03
Glue / IO / Test	2,000	-	-

4 Related Work

Our approach of searching for conjunctions of identical constraints generalizes the idea of *matrix models* [14], which are an often-used pattern in constraint modeling.

The method proposed is a special, restricted case of *constraint acquisition* [25], which is the process of finding a constraint network from a training set of positive and negative examples. The learning process operates on a library of allowed constraints, and a resulting solution is a conjunction of constraints from that library, each constraint ranging over a subset of the variables.

The most successful of these systems is the CONACQ system [6], which proposes the use of version space learning to generate models interactively with the user, relying on an underlying SAT model to perform the learning. This is shown to work for binary constraints, but the method breaks down for global constraints over an unlimited number of variables.

In [7], the authors study the problem of determining argument values for global constraints like the `gcc` from example solutions, in the context of timetabling problems. This is similar to the argument creation we describe in Section 2.3.

The more recent work of [21] considers the use of inductive logic programming for finding models for problems given as a set of logic formulas. This can be powerful to find generic, size-independent models for a problem, but again, it is unclear how to deal with a library of given global constraints, which may not have a simple description as logic formulas.

Our dominance check based on meta-data is related to the work described in [11], where they use a theorem prover to find certain implications between constraints for a restricted domain. This does not rely on meta-data provided in the system, but instead would require a very powerful theorem prover to work for a collection of constraints for problems of the size considered here.

Common to all these results is that they have not been evaluated on a large variety of problems, that they consider only a limited number of potential constraints, and that problem sizes have been quite small.

5 Limitations, Future Work and Conclusions

We are currently only considering some 70 constraints in the global constraint catalog in our seeker calls. Many of the missing constraints require additional information (cost matrix, lookup tables) which have to be provided as additional input data to the system. For some problems, such additional data, like a precedence graph, may also express implicit, less regular sequence generators, which define for which variables a constraint should be stated. Extending our input format to allow for such data would drastically increase both the number of constraints that can be considered, as well as the range of application problems that can be modelled.

Most other constraint acquisition systems use both positive and negative examples. The negative examples are used to interactively differentiate between competing models of the system. We currently only use positive examples, but given recent results on global constraint reification [1], we could extend our system to include this functionality.

If we want to provide the functionality we have presented here to end-users, we will have to consider issues of usability and interactivity, allowing the user to filter and change constraint candidates, as well as being able to suggest custom sequence generators tailored to a specific problem.

Ultimately, we are looking for a modeling tool which can analyze samples of different sizes, and generate a generic, size independent model. Building on top of our existing framework, this would require to express both the sequence generator parameters and any additional arguments for constraints in terms of a variable problem size, to produce more compact, iterative code instead of the flat models currently generated.

Exploiting the idea that many highly structured combinatorial problems can be described by a small set of conjunctions of identical global constraints, this paper proposes a Model Seeker for extracting global constraint models from positive sample solutions. It relies on a detailled description of the constraints in terms of meta-data in the global constraint catalog. The system provides promising results on a variety of problems even when working from a limited number of examples.

Acknowledgement

The help of Hakan Kjellerstrand in finding example problems is gratefully acknowledged.

References

1. N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. On the reification of global constraints. Technical Report T2012:02, SICS, 2012.
2. N. Beldiceanu, M. Carlsson, and J. Rampon. Global constraint catalog, 2nd edition (revision a). Technical Report T2012:03, SICS, 2012.
3. N. Beldiceanu and H. Simonis. A constraint seeker: Finding and ranking global constraints from examples. In Jimmy Ho-Man Lee, editor, *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 12–26. Springer, 2011.
4. N. Beldiceanu and H. Simonis. Using the global constraint seeker for learning structured constraint models: a first attempt. In *The 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*, pages 20–34, Perugia, Italy, September 2011.
5. E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning ways for your mathematical plays - Volume 4*. A K Peters/CRC Press, 2nd edition, 2004.
6. C. Bessière, R. Coletta, E. C. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In M. Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2004.
7. C. Bessière, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 23–34. Springer, 2005.
8. C. Bessière, R. Coletta, and T. Petit. Acquiring parameters of implied global constraints. In P. van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 747–751. Springer, 2005.
9. R. C. Bose, S. S. Shrikhande, and E. T. Parker. Further results on the construction of mutually orthogonal latin squares and the falsity of Euler's conjecture. *Canadian Journal of Mathematics*, 12:189–203, 1960.
10. J. Carlier and E. Pinson. An algorithm for solving the job shop problem. *Management Science*, 35:164–176, 1989.
11. J. Charnley, S. Colton, and I. Miguel. Automatic generation of implied constraints. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 73–77. IOS Press, 2006.
12. Konstantinos Drakakis. A review of Costas arrays. *Journal of Applied Mathematics*, pages 1–32, 2006.
13. H.E. Dudeney. *Amusements in Mathematics*. Dover, New York, 1917.
14. Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Matrix modelling. Technical Report 2001-023, Department of Information Technology, Uppsala University, September 2001.
15. T.W. Haynes, S.T. Hedetniemi, and P.J. Slater. *Fundamentals of Domination in Graphs*. Monographs and Textbooks in Pure and Applied Mathematics. Marcel Dekker, 1998.
16. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, Boston, MA, 2005.
17. M. Henz. Scheduling a major college basketball conference - revisited. *Operations Research*, 49:163–168, Jan/Feb 2001.
18. M. Henz, T. Müller, and S. Thiel. Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, 153(1):92–101, 2004.
19. B. M. Hernández. *The Systematic Generation of Channelled Models in Constraint Satisfaction*. PhD thesis, University of York, York, YO10 5DD, UK, Department of Computer Science, 2007.

20. J. N. Hooker. *Integrated Methods for Optimization*. Springer Science + Business Media, LLC, New York, 2007.
21. Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *ICTAI (1)*, pages 45–52. IEEE Computer Society, 2010.
22. M. J. Maher. Open Constraints in a Boundable World. In W.-J. van Hoeve and J. N. Hooker, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *LNCS*, pages 163–177, Pittsburgh, PA, USA, 2009. Springer-Verlag.
23. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
24. G. Nemhauser and M. Trick. Scheduling a major college basketball conference. *Operations Research*, 46:1–8, Jan 1998.
25. B. O'Sullivan. Automated modelling and solving in constraint programming. In M. Fox and D. Poole, editors, *AAAI*, pages 1493–1497. AAAI Press, 2010.
26. M. S. Petkovic. *Famous Puzzles of Great Mathematicians*. American Mathematical Society, Providence, Rhode Island, 2009.
27. O. Roussel and C. Lecoutre. XML representation of constraint networks format XCSP 2.1. Technical Report arXiv:0902.2362v1, Universite Lille-Nord de France, Artois, 2009.
28. J. A. M. Schreuder. Combinatorial aspects of construction of competition Dutch professional football leagues. *Discrete Applied Mathematics*, 35(3):301–312, 1992.
29. Barbara M. Smith, Sally C. Brailsford, Peter M. Hubbard, and H. Paul Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1(1/2):119–138, 1996.
30. J. P. Walser. *Domain-Independent Local Search for Linear Integer Optimization*. PhD thesis, Technical Faculty of the University des Saarlandes, Saarbruecken, Germany, October 1998.
31. J.J. Watkins. *Across the Board: The Mathematics of Chessboard Problems*. Princeton University Press, Princeton, NJ, 2004.