Globalizing Constraint Models*

Kevin Leo[1], Christopher Mears[1], Guido Tack[12], and Maria Garcia de la Banda[12]

[1] Faculty of IT, Monash University, Australia
[2] National ICT Australia (NICTA), Victoria Laboratory
{kevin.leo,chris.mears,guido.tack,maria.garciadelabanda}@monash.edu

Abstract. We present a method that, given a constraint model, suggests global constraints to replace parts of it. This helps non-expert users to write higher-level models that are easier to reason about and may result in better solving performance. Our method exploits the *structure* of the model by considering combinations of the constraints, collections of variables, parameters and loops already present in the model, as well as parameter data from several data files. We assign a score to a candidate global constraint by comparing a sample of its solution space with that of the part of the model it is intended to replace. The top-scoring global constraints are presented to the user through an interactive display, which shows how they could be incorporated into the model. The *MiniZinc Globalizer*, our implementation of the method for the MiniZinc modelling language, is available on the web.

1 Introduction

Constraint problems can usually be modelled in many different ways, and the choice of model can have a significant impact on the effectiveness of the resulting constraint program. Developing good models is often a very challenging iterative process that requires considerable levels of expertise and consumes significant amounts of resources. This paper introduces a method that supports users through this iterative process: given a constraint problem model and a few input data files, the method suggests global constraints as possible replacements for certain sets of constraints in the model.

Replacing simpler constraints by global constraints — *"globalizing" the model* — has three significant advantages. First, many solvers implement specialised algorithms for global constraints. Therefore, having the global constraint in the model can improve the efficiency of the solving process considerably. Second, more information is made available regarding the underlying structure of the model. The additional information can help, for example, to detect symmetries, which can then be broken either by adding symmetry breaking constraints or by modifying the search. As another example, even if the chosen solver does not yet support the inferred global constraint, its presence in the model can be used to select better decompositions than the ones originally used by the modeller. And third, the higher-level model obtained by the globalization may improve the modeller's understanding of the problem and even make it more readable.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This research was partly sponsored by the Australian Research Council grant DP110102258.

Our method is based on *splitting* a constraint model into submodels, *generating candidate* global constraints for each submodel, and *ranking and filtering* these candidates to produce the output returned to the user. Critically, each of these steps makes extensive use of the existing structure in the model, such as loops and collections of variables, as well as the provided instance data. Note that the correctness of replacing constraints in the model by the candidate global constraints needs to be determined by the user. This approach is similar to that successfully used for symmetry detection [10], which analyses several small instances of a model (i.e., several combinations of model with input data) to obtain candidate symmetries, and then lifts this information from the instances to the model itself.

Our method has many novel characteristics when compared to other automatic model transformation methods (e.g., [6,8,9,7,4,5,2,1]; see Section 6 for a detailed discussion). First, other methods focus on directly inferring a combination of constraints for the entire model, rather than on splitting it into submodels. Splitting allows us to directly associate the candidate global constraint with the group of constraints it replaces (those in the submodel). Second, the generation of arguments for the candidate global constraints uses the variables, parameters and collections of variables appearing in the associated submodel. This allows us to generate likely constraint arguments efficiently. Further, it means the candidate global constraints are defined at the model level rather than at the instance level. This is important not only for the user, but also for our third novel characteristic: our method uses the solutions from different instances (rather than from a single one) to generate, rank and filter the candidates. This increases its accuracy considerably (as shown experimentally in Section 5).

We have implemented the method for the MiniZinc modelling language [11]. The resulting tool – the *MiniZinc Globalizer* – can be accessed through a web interface at `http://www.minizinc.org/globalizer/`. The presented techniques are however not specific to MiniZinc and apply to any representation of a constraint model.

2 Background

We distinguish between *constraint problems*, *models*, and *instances*. A constraint (satisfaction or optimization) problem is the abstract problem we want to solve, e.g., the Graph-colouring problem. A model is a concrete specification of the problem in terms of variables, domains, constraints, and parameters. For the Graph-colouring problem, a model could have variables representing the nodes, domains representing the colours, parameters for the graph and number of colours used, and constraints stating that no two connected nodes can have the same colour. A model together with one concrete set of input data – such as a concrete graph and set of colours – is an instance.

All models used herein are written in MiniZinc. A MiniZinc model consists of a list of variable declarations, parameter declarations, and constraints, as well as a *solve item* that may specify an objective function. The subset of MiniZinc used in this paper should be mostly self-explanatory.

```
1  int: p; int: nh; int: ng;
2  set of int : HostBoats = 1..nh;
3  set of int : GuestCrews = 1..ng;
4  set of int : Time = 1..p;
5  array [GuestCrews] of int : crew;
6  array [HostBoats]  of int : capacity;
7
8  array [GuestCrews, Time] of var HostBoats : hostedBy;
9  array [GuestCrews, HostBoats, Time] of var 0..1 : visits;
10 constraint forall (g in GuestCrews, h in HostBoats, t in Time)
11   (visits[g,h,t] = 1 <-> hostedBy[g,t]=h);                  % channel
12
13 constraint forall (h in HostBoats)
14  ( forall (g in GuestCrews)
15      (sum (t in Time) (visits[g,h,t]) <= 1)                 % distinct_visits
16 /\ forall (t in Time)
17      (sum (g in GuestCrews) (crew[g]*visits[g,h,t]) <= capacity[h]));
18                                                             % capacity
19
20 array [GuestCrews, GuestCrews, Time] of var 0..1 : meet;
21 constraint forall (k, l in GuestCrews where k<l) (
22     forall (t in Time)
23       (hostedBy[k,t] = hostedBy[l,t] -> meet[k,l,t] = 1)    % will_meet
24 /\  sum (t in Time) (meet[k,l,t]) <= 1 );                   % meet_once
```

Fig. 1. A Progressive Party model in MiniZinc

Running example: The Progressive Party problem

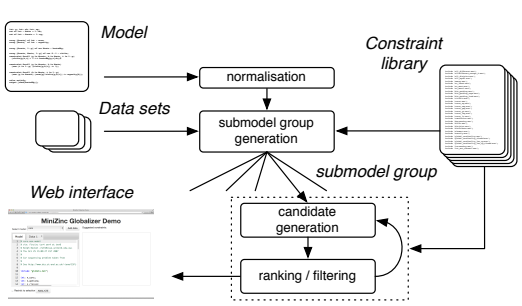
Throughout the paper, we will use a version of the *Progressive Party Problem* as a running example. This problem can be described as follows: to organise a party at a yacht club, certain boats are designated as hosts, while the crews of the remaining boats in turn visit the host boats for several successive fixed-time periods. Every boat has a given maximum capacity for hosting guests, a guest crew cannot revisit a host, and guest crews cannot meet more than once.

As shown in [12], the first known model for this problem was a zero-one integer program by the University of Southampton. Since this model introduced a huge number of constraints, an alternative one was given [12] which found a 13-host solution.

A MiniZinc version of the second model is shown in Fig. 1. Lines 1–6 introduce the parameters: number of time periods, host boats, and guest crews, as well as the sets of designated host boats and guest crews. The main decision variables (line 8) express that at time `t`, guest crew `g` is hosted by boat `hostedBy[g,t]`. Lines 9–11 introduce auxiliary zero-one variables `visits[g,h,t]` that are 1 if and only if `hostedBy[g,t]=h`. These variables are used in line 15 to express that each guest crew visits each host boat at most once; and in line 17 to model the capacity constraints. Finally, lines 20–24 model that guest crews can meet at most once.

The expert modeller can immediately see that line 15 expresses an *alldifferent* constraint on the `hostedBy` variables for each `g in GuestCrews`. The fact that line 17 can be expressed using a set of *bin packing* constraints is slightly less obvious.

To simplify the discussion of our running example, the remaining sections will use the following shorthand notation to express the main structure of the above model: $(\forall GHT : \mathit{channel}) \wedge (\forall H : (\forall G : \mathit{distinct\_visits}) \wedge (\forall T : \mathit{capacity})) \wedge (\forall GG : (\forall T : \mathit{will\_meet}) \wedge \mathit{meet\_once})$, where *channel* denotes the constraint appearing in lines 10–



globalize$(M, Data, Lib)$ =
  $(Constrs, Decls) \leftarrow$ **normalize**(M)
  $Groups \leftarrow$ **generate_groups**$(Constrs, Decls, Data)$
  $Candidates \leftarrow \emptyset$
  for each $Gr \in Groups$:
    $Candidates \leftarrow Candidates \cup$ **process_group**(Gr, Lib)
  return $Candidates$

Fig. 2. An overview of model globalization

11, *distinct_visits* that in line 15, *capacity* that in line 17, *will_meet* that in line 23 and *meet_once* that in line 24. Universal quantifications over G, H and T correspond to loops over the sets `GuestCrews`, `HostBoats`, and `Time`, respectively. We call G, H and T the *index sets* of their loops. For simplicity, we always write nested `forall` loops using a single quantifier and disregard the order of their index sets, e.g., $\forall GHT$ is equivalent to $\forall T\,\forall G \forall H$, to $\forall G \forall T\,\forall H$, and so on.

3 Globalization

Figure 2 provides a graphical and algorithmic view of the main steps of our method, which are as follows. First, the input model file M together with one or more data files *Data* are read. Then, M is *normalized* by splitting conjoined constraints and separating the constraints *Constrs* from the variable and parameter declarations *Decls*. After normalization, several *submodel instance groups* are generated, where each such group $Gr \in Groups$ corresponds to the instantiation of a single submodel with each of the data files in *Data*. A submodel is formed by the combination of *Decls* with a subset of *Constrs*. For each group *Gr*, the method generates a set of *candidate global constraints* from those present in the constraint library *Lib*, where each global constraint in this set is a candidate for equivalence to the submodel associated to *Gr*. The generated candidates are then *scored* according to how well their solution space matches that of the submodel, and are *filtered out* if their score is below a given threshold. Finally, the resulting candidates are shown to the user by means of an interactive GUI. The following sections discuss each of these steps in detail.

3.1 Generating submodel instance groups

The algorithms for normalizing a model M and generating its submodel instance groups *Groups* are shown in Fig. 3. The **normalize** procedure partitions M into two sets: the set *Constrs* of *normalized* constraints and the set *Decls* of original variable and parameter declarations. Constraints are normalized by exhaustively applying two rewriting rules that (a) turn top-level conjunctions into individual constraints and (b) split `forall` loops that contain conjunctions into individual `forall` loops. For example,

normalize(M)
  $Constrs \leftarrow$ set of constraints in M
  while one of the following rules applies:
    if there is a $c \in Constrs$ of the form $(c_1 \wedge \ldots \wedge c_n)$:
      replace c with c_i for each i
    if there is a $c \in Constrs$ of the form $(\forall A_1 \ldots \forall A_n : c_1 \wedge \ldots \wedge c_m)$:
      replace c with $(\forall A_1 \ldots \forall A_n : c_i)$ for each i
  $Decls \leftarrow$ set of all variable and parameter declarations in M
  return ($Constrs$,$Decls$)

generate_groups($Constrs$,$Decls$,$Data$)
  $Groups \leftarrow \emptyset$
  for each $SC \subseteq Constrs$ such that SC is connected:
    $Groups \leftarrow Groups \cup \{$ **instantiate**$(\emptyset, SC, Decls, Data)$ $\}$
      $\cup$ **unroll_loops**$(\emptyset, SC, Decls, Data)$
  return $Groups$

unroll_loops($Fix, SC, Decls, Data$)
  $Groups \leftarrow \emptyset$
  for each A such that $(\forall \ldots A \ldots : d)$ is in all $c \in SC$:
    for each combination L of loops $\forall A$, one for each $c \in SC$:
      $Groups \leftarrow Groups \cup \{$ **instantiate**$(\{L\} \cup Fix, SC \setminus L, Decls, Data)$ $\}$
        $\cup$ **unroll_loops**$(\{L\} \cup Fix, SC \setminus L, Decls, Data)$
  return $Groups$

instantiate($Fix, SC, Decls, Data$)
  $Gr \leftarrow \emptyset$
  for all combinations of min, max for all $L \in Fix$ and all $D \in Data$
    create submodel instance SI from submodel $(SC \cup Decls)$
    $Gr \leftarrow Gr \cup \{SI\}$
  return Gr

Fig. 3. Splitting a model M into groups of submodel instances

after normalizing the Progressive Party model, *Constrs* will contain the following five constraints $\forall GHT$: *channel*, $\forall GH$: *distinct_visits*, $\forall HT$: *capacity*, $\forall GGT$: *will_meet* and $\forall GG$: *meet_once*.

Normalization is vital for discovering global constraints that describe parts of a top level constraint. For example, we said that the combination of constraint $\forall GH$: *distinct_visits* with channelling constraint $\forall GHT$: *channel* in the Progressive Party is equivalent to a conjunction of `alldifferent`s. To discover this, we need to consider each component constraint separately, so that we can combine them appropriately.

Once normalization is complete, **generate_groups** produces every *connected* subset of constraints in *Constrs*.[3] Two constraints are connected if they share at least one variable. A set of constraints is connected if for each pair of constraints c, c' in the set, a path of constraints can be found starting with c and ending with c', such that consecutive

[3] Our implementation has a parameter to limit the maximum size of the generated subsets (the default is 3 as our experiments have not found globals from larger conjunctions of constraints).

constraints on the path are connected. For example, the subset of normalized constraints $SC = \{\forall GHT : channel, \forall GG : meet\_once\}$ in the Progressive Party is not connected, since their sets of variables are $\{$`visits`, `hostedBy`$\}$ and $\{$`meet`$\}$, respectively. Since the set of global constraints inferred for a non-connected *SC* would at best be identical to the union of those found for each of its constraints separately, we can discard *SC*.

Every connected subset $SC \in Constrs$ is passed to **instantiate**, together with the set of declarations *Decls* and the input data files *Data*, to create a group of related submodel instances: one per combination of the submodel $SC \cup Decls$ with a data file *D* in *Data*.

Considering all connected subsets of the normalized top-level constraints is, however, not enough. This can be illustrated with the Progressive Party model: since no global constraint describes a conjunction of `alldifferent` constraints, the normalized constraints $\forall GH : distinct\_visits$ and $\forall GHT : channel$ must be combined, unrolled and instantiated in such a way as to make them discoverable. Loop unrolling achieves this by recursively combining and instantiating the loops in each *SC* subset of *Constrs* as follows. For each index set *A* that appears in every constraint of *SC*, it computes each *combination L* of `forall` loops over *A*, choosing one from each constraint $c \in SC$. Let us explain how such a combination is computed for subset $SC = \{\forall GHT : channel, \forall GH : distinct\_visits, \forall GGT : will\_meet, \forall GG : meet\_once\}$ and loop *G* (which appears in every constraint of *SC*). We first label the individual loops to be able to identify them: $\forall G_1HT : channel$, $\forall G_2H : distinct\_visits$, $\forall G_3G_4T : will\_meet$, $\forall G_5G_6 : meet\_once$. We then compute all combinations of *G* that have one index set from each constraint, obtaining 4 combinations: $L_1 = \{\forall G_1, \forall G_2, \forall G_3, \forall G_5\}$, $L_2 = \{\forall G_1, \forall G_2, \forall G_3, \forall G_6\}$, $L_3 = \{\forall G_1, \forall G_2, \forall G_4, \forall G_5\}$, and $L_4 = \{\forall G_1, \forall G_2, \forall G_4, \forall G_6\}$.

For each combination *L*, loop unrolling calls **instantiate** with *L* added to the set of combinations *Fix* to be *fixed*, and removed from *SC*. Fixing a loop means instantiating its index variable to a particular value from its index set. Our algorithm fixes indices to two values: the minima and maxima of their index set. Consider, for example, index set *G* and $SC = \{\forall G_1HT : channel, \forall G_2H : distinct\_visits\}$. The only combination *L* for *G* is $L = \{\forall G_1, \forall G_2\}$. Thus, **instantiate** will fix the index variable *g* of G_1 and G_2 to the same value resulting in the following two submodels:

$$(g = \min(G)) \wedge (\forall HT : channel) \wedge (\forall H : distinct\_visits)$$
$$(g = \max(G)) \wedge (\forall HT : channel) \wedge (\forall H : distinct\_visits)$$

each describing an `alldifferent` constraint over the variables `HostedBy[g,h]`, for a fixed value of `g`. Both submodels are then instantiated with any provided data, resulting in submodel instances that are added to the same group. Each will also be submitted to further loop unrolling leading to the following submodels:

$$(g = \min(G) \wedge h = \min(H)) \wedge (\forall T : channel) \wedge (distinct\_visits)$$
$$(g = \min(G) \wedge h = \max(H)) \wedge (\forall T : channel) \wedge (distinct\_visits)$$
$$(g = \max(G) \wedge h = \min(H)) \wedge (\forall T : channel) \wedge (distinct\_visits)$$
$$(g = \max(G) \wedge h = \max(H)) \wedge (\forall T : channel) \wedge (distinct\_visits)$$

This process is repeated recursively until all loops have been unrolled.

A special case to be considered is what the algorithm should do upon fixing all the loops in a constraint when the original `forall` loop had a `where` clause. During

```
generate_candidates(SI,Decls, Lib)=
    Candidates ← ∅
    Solutions ← random sample of solutions of submodel instance SI
    Template ← (SI \ Decls) ∪ Solutions
    Base_arguments ←
            (variable and parameter collections in SI) ∪
            (variable and parameter sub-collections in constraints of SI) ∪
    Arguments ← Base_arguments ∪
            (array accesses of elements of Base_arguments) ∪
            { constant 0 } ∪
            { blank symbol }
    for each constraint cons in Lib:
        for each tuple args that can be built from Arguments:
            Replace blank symbols in args by their value
            Instance ← Template ∪ (constraints for cons(args))
            if Instance is satisfiable
                add cons(args) to Candidates
    return Candidates
```

Fig. 4. Generating candidate constraints for a submodel instance.

unrolling, and as long as there is one `forall` the algorithm leaves the `where` clause as part of that `forall`, but when there is no `forall` left, the resulting constraint gets wrapped in an `if-then-else` expression to avoid creating incorrect submodels.

3.2 Candidate generation

As explained in the previous section, once **generate_groups** finishes, *Groups* has all submodel instance groups, where each group contains different instances of the same submodel. Recall that each instance has different parameter values due either to different data files given by the user, or to the different minima and maxima values chosen during loop unrolling. Each group *Gr* in *Groups* is then processed to generate a set of candidate constraints, which are added to the final set *Candidates*. The algorithm for generating candidate constraints for each submodel instance $SI \in Gr$, given the set *Decls* of declarations of the original model *M*, and the library of global constraints *Lib*, is shown in Fig. 4. Note that each constraint entry in *Lib* has a name, arity, and type of arguments. In addition, arguments can have associated information indicating whether they are functionally dependent on other arguments, and stating conditions that must be met for the argument to be used. See Section 4.2 for details on the particular *Lib* used by our implementation.

The algorithm proceeds as follows. After finding a random sample of solutions of *SI*, we build a template model by replacing the parameters and variables in *Decls* by the sample solutions. The template includes *all* the sample solutions, as we want the candidate global constraint to satisfy all of these sample solutions: a single sample solution that violates the constraint is sufficient evidence to discard the constraint. This template model is trivially satisfiable. Intuitively, a global constraint will be considered as a candidate if it is satisfied by the sample solutions of *SI* — that is, if after adding the candidate, the template model remains satisfiable.

Candidate constraints are obtained by combining each constraint cons in *Lib* with an *A*-tuple args of arguments, where *A* is the arity of the constraint. The arguments are drawn from the identifiers that appear in *SI*. These include the variable and parameter collections whose identifiers appear in *SI*, those same collections restricted to their subsets that are actually used in the constraints of *SI*, array access expressions composed from the two previous groups (referred to as base_arguments), the constant zero, and a special blank symbol. This blank symbol is used as a place-holder for arguments known to be functionally defined by the others. Once all non-blank arguments are selected, the blank symbol is replaced by its corresponding value. Note that this value must be the same for all sample solutions. If the constraint is not functional, or if the sample solutions disagree on what the value should be, args is discarded. Functionally-defined arguments are further required to take the same value *across all instances*. This is however not a significant issue, as they are only used when no named parameter is found.

Finally, the candidate global constraint cons(args) is added to the template model and the resulting model is evaluated. If the constraint holds, cons(args) is added to the list of candidate global constraints for *SI*.

Let us illustrate this process with the subinstance *SI* formed by combining the constraint $(g = \min(G)) \wedge (\forall HT : \mathit{channel})$ of the Progressive Party model (where $\min(G) = 1$) with the variable and parameter declarations in the model, and some data file $D \in \mathit{Data}$. The base arguments for *SI* include the variable collections hostedBy and visits, the variable sub-collections hostedBy[1,t] and visits[1,h,t], and the parameter collections HostBoats, GuestCrews, p, nh, ng, Time, and the index g itself (with value 1). The arguments are the constant 0, the blank symbol, the base arguments, and array accesses formed by combining an array with a parameter, e.g., crew[nh] and hostedBy[p,capacity]. After considering all constraints in *Lib* with these arguments, the following candidate constraints are generated (among many others):

- lex2(hostedBy)
- alldifferent([hostedBy[g,1],hostedBy[g,2],...,hostedBy[g,p]])
- sliding_sum(0, p, g, [hostedBy[g,1],hostedBy[g,2],...,hostedBy[g,p]])

Note that in the first constraint the entire hostedBy array is used as an argument, while in the second and third constraints only that subset of the array that participates in the constraints of the submodel – where g is fixed to 1 – is used as an argument.

An alternative to this form of candidate generation is to syntactically match groups of constraints to known (correct) reformulations. We do not take this approach as it would be much too restrictive, requiring the modeler to have implemented exactly the constraints we are looking for.

3.3 Ranking and filtering

As shown in Figure 5, submodel instances are processed in groups, where each group *Gr* contains submodel instances of the same model. This allows us to accurately determine the candidate constraints for *Gr* by taking the intersection of the candidate constraints found for each submodel instance *SI* in *Gr*. For the first *SI* being processed, the full set of constraints and argument tuples (written as the special symbol *Universe*)

```
process_group(Gr,Decls,Lib)=
    Candidates ← Universe
    for each submodel instance SI in group Gr:
        Candidates ← Candidates ∩ generate_candidates(SI,Decls,Lib)
        for each candidate constraint cons(args) in Candidates:
            SolutionsC ← random sample of solutions of cons(args)
            SolutionsM ← subset of SolutionsC that are also solutions of SI
            if |SolutionsM| ÷ |SolutionsC| < equivalenceThreshold
                Delete cons(args) from Candidates
                for each possible context B:
                    SolutionsC ← random sample of solutions of cons(args) ∧ B
                    SolutionsM ← subset of SolutionsC that are also solutions of SI
                    if |SolutionsM| ÷ |SolutionsC| ≥ equivalenceThreshold
                        Add cons(args) with context B to Candidates
    return Candidates
```

Fig. 5. Ranking and filtering constraints.

is considered, so that the intersection is the candidate set generated for this *SI*. Due to filtering the set decreases for subsequent instances in the group and, after processing the final one, the remaining candidates are exactly the intersection we seek to compute.

For each candidate constraint cons(args) inferred for a given *SI*, **process_group** measures how closely it matches *SI*. To achieve this, we collect a random sample of solutions of cons(args), and compute the fraction of these solutions that are also solutions to *SI*. If the constraint is equivalent to the submodel of *SI*, this fraction must be 1; if the constraint is a poor match, the fraction should be close to 0. We filter the candidates by keeping only those constraints whose matching fraction is greater than a given threshold. A threshold of 0.5 has been shown experimentally to be sufficient to eliminate imperfect matches, and we use that value in our implementation.

In some cases a constraint in a model is equivalent to a candidate global constraint only in the context of another constraint. Consider a submodel instance *SI* containing constraints *A* and *B*, and a candidate global constraint cons(args), where *A* is not equivalent to cons(args), but the conjunction $A \wedge B$ is equivalent to the conjunction cons(args)$\wedge B$. We call *B* the *context* in which *A* is equivalent to cons(args). For example, let *SI* have the constraints $(t = \min(1..p)) \wedge \forall GH : channel \wedge \forall H : capacity$ from the Progressive Party. The global constraint bin_packing_capa(capacity,hostedBy[1..ng,t], crew) is equivalent to $\forall H : capacity$, but only in the context of $\forall GH : channel$. In general, a contextually-equivalent constraint cons(args) will be implied by *SI* but appear weaker than the instance and, thus, will score badly during ranking. In this case, we try using one of *SI*'s constraints as the context constraint *B*, and test via sampling whether cons(args)$\wedge B$ implies the submodel instance. If this scores well, we say that *A* is equivalent to cons(args) under the context of *B*, and add this to the list of candidates.

For the purpose of scoring and filtering, a candidate constraint should now be considered a pair of the cons(args) and its context. Note that the context may be empty. This means that for a context-dependent constraint to pass the filtering tests, it must pass with the same context in all instances of the group.

Table 1. Library of global constraints

alldifferent	circuit	global_cardinality	nvalue
alldifferent_except_0	count	increasing	sliding_sum
all_equal	cumulative	inverse	sort
atleast	decreasing	lex_less	strict_lex2
atmost	diffn	lex_lesseq	subcircuit
bin_packing	distribute	lex2	unary
bin_packing_capa	element	maximum	value_precede
bin_packing_load	exactly	minimum	
channel	gcc	member	

4 Implementation

We have implemented the globalization system for MiniZinc models. We use the libmzn C++ library for parsing and manipulating MiniZinc model and data files. The model evaluator is written in Haskell, and uses bindings to call libmzn.

4.1 Checking versus solving

As shown before, when generating the candidates of a given submodel instance *SI* of group *Gr*, our method first solves *SI* to find a random sample of its solutions. To do this, our implementation uses the standard MiniZinc tool mzn2fzn to flatten *SI*, and the Gecode constraint solver to find 30 random solutions for it. Theses are found with a search that selects values in random order, and restarts from scratch whenever a solution is found. If the search is not complete within 60 seconds, *SI* is discarded.

Later in the process our method checks the satisfiability of the instance resulting from adding to the template the possible candidate constraints. Since this template has no variables, and the added constraints are simply evaluated, no search is required. Such checks are performed very often, and the expense of flattening the instance and calling a full constraint solver is crippling. To avoid this, we have implemented a simple evaluator of MiniZinc instances known not to have variables. In practice, this optimization is crucial, as the number of evaluations is usually in the hundreds of thousands.

4.2 Library of global constraints

Table 1 lists the global constraints in our implementation of *Lib*, which is used for candidate generation. These are all the global constraints defined in MiniZinc's standard library (version 1.6) over integer arguments (sets are not handled yet by our prototype implementation), with the addition of the following constraints:

- channel(x,a): channels an integer variable x to an array of 0-1 variables a.
- gcc(x,counts): a special case of global_cardinality where the "cover" argument, which specifies a map from indices to values, is fixed to the identity map.
- unary(s,d): a special case of cumulative where the resource capacity and the usage for each task are fixed to 1, implementing a unary resource constraint.

We are able to evaluate most of the constraints using the default decomposition given in the MiniZinc library. However, some decompositions introduce variables which are not handled by our simple satisfiability check evaluator (recall that our satisfiability check does not perform search). Thus, in these cases we evaluate the constraint directly.

As mentioned in Section 3.3, each constraint is annotated with conditions for its use to prevent the constraint being considered as a candidate when it is trivially true or otherwise useless. For example, the alldifferent constraint specifies that its argument must be an array of variables with arity greater than one since, otherwise, the constraint is trivially true or nonsensical. As another example, the sliding_sum(l,u,n,x) constraint specifies that every n-length subsequence of x must sum to a value between l and u. When generating candidates, we ensure that $l < u$, $1 < n <$ length(x), and $l > n \times$ lb_array(x) $\vee$ $u < n \times$ ub_array(x). The first two conditions ensure that the parameters make sense, while the third one ensures that the constraint is tighter than what is already imposed by the domains of the variables in x. This last condition is added for efficiency reasons, as the ranking and filtering process would have taken care of it.

4.3 Web interface

The MiniZinc Globalizer is implemented as an asynchronous web server that queues the requests made by clients and can execute several requests in parallel. Requests can be cancelled by the user and are automatically cancelled when the session is terminated, for instance, when a user closes the browser window. The Globalizer is publicly available at `http://www.minizinc.org/globalizer/`.

Figure 6 shows a screen shot of the web interface. Users can enter their model and instance data through an embedded editor (seen on the left). Clicking ANALYZE launches the request, initiating a progress bar that provides the user periodic progress updates. When the analysis finishes, the right hand side of the window displays the results. Clicking on any candidate constraint highlights in yellow the part of the original model that the constraint could replace, and highlights in orange the candidate's context, if any. The interface allows the user to select parts of the model and restrict the analysis to the selected parts by selecting "Only selection" in the lower left corner of the window. The analysis will then only use the selected constraints. This is useful when the analysis is taking a long time, or the user wants to focus on a particular part of the model.

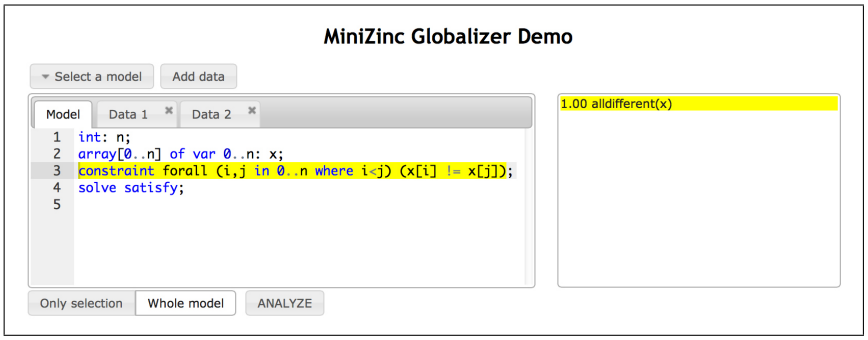


Fig. 6. The web interface to the MiniZinc Globalizer

Table 2. Experimental Results

problem	time	$\|Groups\|$	calls	evals	top candidates
Cars	215	12	2910	31335	**gcc(step_class,cars_in_class)** count(step_class,c,cars_in_class[c]) **sliding_sum(sliding_sum(0,option_max_per_block[p],** **option_block_size[p],step_option_use[1..10,*])**
Jobshop	23	16	410	3620	**unary(s[1..n,*], d[1..n,*])**
Party	691	48	4214	36429	**bin_packing_capa(spareCapacity,hostedBy[i..n,*],crew)** **alldifferent(hostedBy[*,1..4])** **channel(hostedBy[*,*], visits[*,1..4,*])** unary(hostedBy[*,1..4],visits[*,*,1..4])
Packing	1659	29	32174	114597	**diffn(x,y,pack_s,pack_s)** **diffn(y,x,pack_s,pack_s)**
Schedule	174	13	3077	22835	**gcc(x,[1,0,1,0,1,0,1])**
Sudoku 1	3996	166	6305	24465	**alldifferent(p[*,1..9])** gcc(p[*,1..9], [1,1,1,1,1,1,1,1,1]) **alldifferent(p[1..9,*])** gcc(p[1..9,*], [1,1,1,1,1,1,1,1,1])
Sudoku 2	335	7	338	2347	-
Warehouses	245	24	3853	69871	**gcc(supplier,use)**

5 Experiments

This section evaluates the accuracy and practicality of the prototype implementation of our MiniZinc Globalizer. The evaluation is performed over a set of constraint problems, each with a number of different data sets. The MiniZinc models used for these problems are available at the MiniZinc Globalizer website.

The results are shown in Table 2, where for each problem model we show: the name of the problem (**problem**), the time in seconds to run the MiniZinc Globalizer (**time**), the number of submodel instance groups obtained ($|Groups|$), the number of calls to Gecode to obtain sample solutions of either submodel instances or global constraint candidates (**calls**), the number of satisfiability tests performed (**evals**), and the global constraints proposed as candidates with score 1 (**top candidates**), where high quality candidates appear in bold. We have manually simplified the output, and excluded some duplicate constraints where the system was unable to distinguish two parameters that appear different, but actually refer to the same value. The set of problems used in the table is as follows.

Cars is a version of the car sequencing problem (CSPLib 001) as implemented in the MiniZinc distribution. It uses simple arithmetic and counting constraints to express the capacity and sequence restrictions of the problem. The Globalizer finds the corresponding `sliding_sum` and `global_cardinality` constraints.

Jobshop is a simple job-shop scheduling problem taken from the MiniZinc distribution. It implements the non-overlapping of two tasks on a unary resource using simple reified constraints. Globalization finds the `unary` scheduling constraint.

Party is our running example from Figure 1. As discussed earlier, the Globalizer finds the `bin packing` and `alldifferent` constraints. It also finds the `channel` global constraint.

Packing packs `n` squares into a rectangle. The source code was taken from the MiniZinc distribution. The Globalizer finds `diffn` constraints that express the non-overlapping of rectangles.

Schedule is a contrived scheduling example from [4]. The schedule is constrained in a way that one task needs to start on every even time point, which implies a global cardinality constraint with argument `[1,0,1,0,...]`. Our analysis can find this constraint as long as all instances have the same schedule length, as otherwise the arguments differ in length between instances and are thus discarded. The generalization of such sequences is left to future work.

Sudoku 1 and 2 are different models for the Sudoku puzzle. The first one uses a zero-one integer linear programming formulation, and the Globalizer finds some `alldifferent` constraints. The second model posts binary not-equal constraints on variables that are organised by row and column, in a complicated set of nested `forall` loops. Here, the loop unrolling is not strong enough to generate candidates that correspond to individual rows, columns, or blocks. As a result, Globalizer cannot find any replacement global constraints.

Warehouses is a warehouse allocation problem (CSPLib 034), whose source code was taken from the MiniZinc distribution. The Globalizer finds that a loop containing counting constraints can be aggregated into one `global cardinality` constraint.

Discussion

The models discussed here are taken either from the literature or from the example suite that comes with MiniZinc. In most cases, the Globalizer has been able to find the global constraints that an expert modeller would have used.

The current prototype is not optimized for performance. The time to analyse a reasonably complex model with 3-4 data sets is in the range of minutes up to an hour, depending mainly on the number of candidates that need to be checked for satisfiability (a number that grows considerably with the number of possible arguments). There is still great potential for improving performance by both avoiding and parallelizing unnecessary candidate checks and parameter instantiations which, as indicated, make up for the bulk of the run time.

The number of generated groups is relatively small (usually less than 50), which means that the problem splitting algorithm achieves a good level of pruning. Generating only a small number of groups and top scoring constraints is important since the results are meant to be presented to a human user.

Looking at the number of satisfiability tests, which can reach hundreds of thousands, it becomes clear that each check needs to be very efficient. This justifies the introduction of a dedicated constraint evaluator as discussed in Section 4.1.

It is interesting to note the effect of using more than one data file for a given problem. For example, analysing the packing problem with a single data file results in 54 candidate global constraints with a score of 1. Adding the additional data files increases the discriminative power of the system by reducing the candidates with a score of 1 to the two shown in Table 2. Similarly, analysing Warehouses with only one data file results in 6 candidate global constraints with a score of 1, as opposed to one as in the table above. For the Schedule and Jobshop problems, however, a single data file was enough to narrow the candidates down to a single constraint with a score of 1.

6 Related work

There are two main lines of research related to this work: *constraint acquisition* and *automatic model transformation*.

In the *acquisition* line of work, Constraint Seeker [1] infers global constraints from positive and negative examples of solutions, and Model Seeker [2] infers an entire model (i.e., conjunctions of constraints) from complete solutions to a constraint problem. This differs from the method presented in this paper both in motivation and methodology. Our motivation is to identify parts of a given model that can be replaced by global constraints. Having access to an initial model significantly affects our methodology, as it allows us to make extensive use of the information contained in the model. In particular, it allows us to (a) focus on submodels that are equivalent to a single global constraint, as opposed to a conjunction of them, (b) significantly reduce the search for possible combinations of global constraint arguments, while increasing the likelihood of obtaining meaningful ones, and (c) consider not only the solution variables, but any other intermediate variables in the model and its input data. Having the input data also affects our methodology, as it allows us to (a) better generate candidates and (b) automatically generate as many solutions as we require for our rankings. For example, the input data enables us to derive bin packing constraints for the Progressive Party problem, while Model Seeker cannot infer these from just the solutions.

Note that we could use Model Seeker to generate more complex candidates for each submodel, and Constraint Seeker to infer and rank candidate constraints. We would like to experimentally evaluate and compare these approaches to our own submodel and constraint generators when the two tools become publicly available.

Our method is also related to the CGRASS system [6,8], which among other model transformations, includes a specialised component to detect alldifferent global constraints for instances of the problem. The main differences are that our Globalizer aims at inferring any of a set of global constraint using a general (rather than specialised) method, and does so for a model, rather than for each of its instances.

Other acquisition approaches focus on the automatic generation of implied constraints. A general method is described in [5], where machine learning is used to induce constraints for the solutions for small problems, and a theorem prover is then used to show the constraints hold for the model. The generality of the method results in applicability restrictions: the model data can only be a single integer, and the model needs to be expressible in first order logic. In our case the constraints are already pre-determined (the list of global constraints considered) and, thus, the data can be as complex as necessary. Further, we do not attempt to prove the correctness of the constraints as this reduces to proving the equivalence of two models, which is undecidable.

Another related method is that of CONACQ [3] which, given examples of solutions and non-solutions for a target problem and a library of constraints, *acquires* constraint networks, that is, conjunctions of constraints in the library that are consistent with the given solutions and non-solutions. CONACQ uses SAT-based *version space algorithm*, where the version space is the set of all constraint networks defined from the library that are consistent with the examples. While general and powerful, it considers instances of models, rather than models themselves. Further, it relies on the library of constraints being relatively small. This is not the case for our approach. As far as we know, CONACQ

currently only handles binary constraints and is not publicly available. Finally, [4] describes how implied parametric constraints can be learned by adding a large disjunction of constraints with different parameters that together are guaranteed to be implied, and then successively pruning that disjunction by checking if certain sets of parameters can be removed without changing the solution space. This method goes further than what we attempt in that it infers parameters from solutions, while we only try to match parameters that are already given in the model. For functional dependencies, however, we can infer parameter sets, as in the Schedule example in Section 5.

In the area of model transformations, the work on Essence [9,7] is somewhat related. These systems transform a model specified in a highly abstract manner into a more concrete one. Our method moves in the opposite direction: we detect parts of a concrete model that are instances of a more generic model pattern. While currently this generic pattern is restricted to global constraints, it is straightforward to extend the method to use any other useful constraint pattern. In fact, globalization and automatic transformation are complementary: starting from a low-level model, globalization yields a high-level model that is then amenable to automatic transformation.

7 Conclusion

This paper has introduced a method for *globalizing* constraint models. Given a constraint model, the method proposes global constraints to replace parts of it. This helps users improve their models, since global constraints capture the inherent structure of a model and can thus help obtain a better translation to the underlying solving technology and faster solving using specialised algorithms.

The inference process is based on splitting a model into submodels that correspond to subsets of its constraints, potentially unrolling loops, and instantiating each of the resulting submodels with different data sets into a group of submodel instances. From these groups of instances, candidate constraints are generated by sampling the solution space of both the group and the candidate constraints. The candidates are ranked and filtered based on how well their search spaces match.

We have presented experimental evidence that the method is both practical and accurate. Our implementation, the MiniZinc Globalizer, is available as a web-based tool.

Regarding future work, while the system already provides useful results, there are some improvements we are planning to explore. First, we would like to incorporate ranking techniques from Constraint Seeker, such as using known implications between constraints to eliminate more imperfect candidates. Second, in order to make contexts more useful, we need to detect global constraints on alternative viewpoints by automatically introducing channeling constraints (Model Seeker follows a similar approach). Third, we would like to generalise argument sequences (such as the `[1,0,1,0...]` in the simple scheduling example from Section 5) and to detect more complex expressions as constraint arguments. Fourth, the confidence in the suggested constraints may be improved by using theorem proving or other techniques to prove equivalence in cases where it is possible. Finally, we would like to integrate the system into an IDE that lets users refactor models automatically using the suggestions generated by the Globalizer.

References

1. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In: Lee, J.H.M. (ed.) Principles and Practice of Constraint Programming-CP. LNCS, vol. 6876, pp. 12–26. Springer (2011)
2. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Milano, M. (ed.) Principles and Practice of Constraint Programming-CP. LNCS, vol. 7514, pp. 141–157. Springer (2012)
3. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: A SAT-based version space algorithm for acquiring constraint satisfaction problems. In: Machine Learning: ECML 2005, pp. 23–34. Springer (2005)
4. Bessiere, C., Coletta, R., Petit, T.: Learning implied global constraints. In: Veloso, M.M. (ed.) IJCAI. pp. 44–49 (2007)
5. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: European Conference on Artificial Intelligence-ECAI. vol. 141, pp. 73–77. IOS Press (2006)
6. Frisch, A., Miguel, I., Walsh, T.: Extensions to proof planning for generating implied constraints. In: Calculemus-01 (2001)
7. Frisch, A.M., Jefferson, C., Martínez-Hernández, B., Miguel, I.: The rules of constraint modelling. In: International Joint Conference on Artificial Intelligence. vol. 19, pp. 109–116. Lawrence Erlbaum Associates LTD (2005)
8. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: A system for transforming constraint satisfaction problems. In: Recent Advances in Constraints, LNCS, vol. 2627, pp. 15–30. Springer (2003)
9. Gent, I.P., Miguel, I., Rendl, A.: Tailoring solver-independent constraint models: A case study with Essence' and Minion. In: Miguel, I., Ruml, W. (eds.) Abstraction, Reformulation, and Approximation, LNCS, vol. 4612, pp. 184–199. Springer (2007)
10. Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B.: A novel approach for detecting symmetries in CSP models. In: Perron, L., Trick, M.A. (eds.) CPAIOR. LNCS, vol. 5015, pp. 158–172. Springer (2008)
11. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming-CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)
12. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: Integer linear programming and constraint programming compared. Constraints 1(1), 119–138 (1996)