Globalizing Constraint Models*

Kevin Leo[a], Christopher Mears[b], Guido Tack[a], Maria Garcia de la Banda[a]

[a]*Faculty of IT, Monash University, Australia*
{kevin.leo,guido.tack,maria.garciadelabanda}@monash.edu
[b]*Redbubble, Melbourne, Australia*
chris.mears@redbubble.com

Abstract

We present a method to detect implicit model patterns (such as global constraints) that might be able to replace parts of a combinatorial problem model that are expressed at a low-level. This can help non-expert users write higher-level models that are easier to reason about and often yield better performance. Our method generates candidate model patterns by analysing both the *structure* of the model – its constraints, variables, parameters and loops – and the input data from one or more data files. Each candidate is scored by comparing a sample of its solution space with that of the part of the model it is intended to replace. The top-scoring candidates are presented to the user through an interactive display, which shows how they could be incorporated into the model. The method is implemented for the *MiniZinc* modelling language and available as part of the MiniZinc distribution.

1. Introduction

Combinatorial problems are solved by finding a combination of choices that satisfies a set of constraints, and possibly optimises an objective function. These decision making problems occur in every area of our lives – from health and transport, to energy and human resources – and finding high quality solutions to these problems is crucial for our society.

Modern approaches for solving combinatorial problems first specify a *model* of the problem that formally describes its choices (in terms of variables and their

*This research was partly sponsored by Australian Research Council grants DP110102258 and DP180100151.

domains), its constraints and its objective function. These models are often specified in *parametric* form, i.e., some variables are defined as *parameters*, thus allowing their values to be provided as input data separate from the model. This allows the same model to be *instantiated* with different input data, as it is required by many applications. For example, a manufacturing company may want to compute optimal schedules for their production lines each day, based on a new set of customers' orders. The combination of model and input data is called an *instance*. To solve a given instance, one must select a constraint *solver* that supports the constraints in the model, compile the instance into a *program* that is suitable for the selected solver, and execute it to find a solution.

The economic impact of solving these kinds of decision problems has fuelled research in powerful modelling languages (e.g., [1, 2, 3, 4]) and solvers (e.g., [5, 6, 7]), which are used by leading businesses such as Amazon, British Steel, Ford, Google, HP, and Nestle, among many others. Importantly, the same problem can be modelled in many different ways and solved using many different solvers, with these differences significantly affecting the time needed to find high-quality solutions. And yet, while there have been significant advances in the variety and quality of solvers available (e.g., Wikipedia now has 125 pages of optimisation algorithms), advances in technology that can help users improve their models have been scarce (see Section 3). Developing such technology is difficult but crucial, as model improvements can yield speed-ups of several orders of magnitude (e.g., [8]). Without it, even expert users must follow an iterative modify-and-test approach that is extremely time consuming and might still yield poor models.

Our aim is to support users through this iterative process, by suggesting possible model improvements based on the detection of *implicit model structures* that can be made explicit by adding them to the model. Consider, for example, the structure provided by *global constraints*, a concept that was developed in the context of Constraint Programming (CP) algorithms [5]. A global constraint expresses a common relation between a non-fixed number of variables, and is often equivalent to the conjunction of several simpler constraints. An example is the global constraint `alldifferent`(xs), which ensures all variables in the array xs are assigned different values. This is logically equivalent to the conjunction of pairwise dis-equalities between all variables in xs. Using global constraints in the model as a shorthand for frequently recurring patterns of simpler constraints, not only simplifies the programming task, but can also improve solving time. This is because most CP solvers use the structure information provided by global constraints to implement dedicated inference algorithms that are more efficient.

Interestingly, using global constraints as *modelling patterns* can be beneficial, even if the solver does not have dedicated inference algorithms for them. This is because global constraints *expose* information regarding the model structure, and this information can be useful to others. For example, the compiler can use it to choose the most efficient decomposition into the simpler constraints supported by the target solver [9]. Analyser tools can use it to improve the automatic detection of symmetries [10], which can in turn be used by symmetry breaking approaches to speed up the solving process. Finally, humans might find global constraints more readable and "self-documenting" than a decomposition into basic constraints. Importantly, these benefits apply to *all instances* of a model – the upfront investment to use global constraints may therefore pay off manifold, with each instance that needs to be solved.

Using global constraints can however be difficult, as naive users might be unaware of global constraints, and expert users might be unfamiliar with the global constraints supported by a specific modelling language. Further, global constraints may require reformulating the model (e.g., if they are only applicable to integer variables, while the model is expressed equivalently in terms of zero-one variables). While these alternative (or dual) *viewpoints* [11] of a model may not be obvious to the modeller, they can also be seen as implicit model structures that, if made explicit, can enable the detection of other implicit structures.

Therefore, we argue that automatic techniques for detecting these implicit structures can be very useful. Since they can be seen, in the widest sense, as global constraints, we call the replacement of a set of constraints by an equivalent global constraint the *"globalization" of the model.*

Main Contribution. This paper presents a method, implemented by the MiniZinc Globalizer system (or Globalizer for short), which given one or more instances of a constraint model, suggests global constraints as possible replacements for *submodels* (subsets of constraints) of the model. For each submodel we generate a set of candidate global constraints. Each candidate is assigned a score based on its similarity to the submodel, where similarity is defined based on sampling the solutions of both the submodel and the candidate. The top-scoring candidates are presented to the user through an interactive display, which shows how they could be incorporated into the model. The method can be applied iteratively, allowing the system to expose alternative viewpoints of the variables in the model.

Our method has many novel characteristics when compared to other automatic constraint model inference and transformation methods (e.g., [12, 13, 14, 15, 16, 17, 18, 19]; see Section 3 for a detailed discussion). First, most other methods fo-

cus on *model acquisition*, i.e., inferring all the constraints for a problem instance from a given sample of solutions and non-solutions of the instance. Instead, our approach starts from an already existing complete and correct model for a problem, and suggests more efficient replacements – candidate global constraints – for some of its submodels. Second, the other methods need to infer constraint parameters and variables from the entire set of instance data and variables. In our method, the generation of arguments for the candidate global constraints uses the variables, parameters and collections (sets/arrays) of variables appearing in the associated submodel. Likely arguments for constraints can then be generated more 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 the third novel characteristic: the method uses solutions from different instances (rather than from a single one) to generate, rank and filter the candidates. This increases its accuracy considerably (as shown in Section 7).

Globalizer is available for download as part of the MiniZinc IDE, which allows for an easy integration into a user's typical modelling workflow.

Structure of the Paper. Section 2 introduces a model that will be used as a running example throughout the paper, together with some common global constraints. Section 3 discusses related work. Section 4 introduces the proposed method to globalize a constraint model, with Section 5 later extending it to infer relationships among hidden structures by representing these hidden structures as constraints to be detected in an initial pass. Section 6 provides some details regarding the implementation of the method used in this paper to perform the experiments reported in Section 7, which show that the method can detect these hidden structures and, in addition, is not prohibitively slow. Finally, Section 8 discusses limitations and future work and Section 9 presents our conclusions.

This article is a extension of [20]. Apart from explaining each step of the presented method in more detail, it contains a more thorough treatment of related work in Section 3, and a significantly extended experimental evaluation in Section 7. Furthermore, this article is based on the version of the Globalizer tool as released with the MiniZinc distribution, whereas the conference paper described the first prototype of the software.

2. Background

This section briefly introduces the MiniZinc modelling language. Note that while our method and its implementation use MiniZinc, they could be adapted for

any other constraint modelling languages that supports global constraints or other high-level structures, such as ESSENCE [1], ESSENCE' [15] or AMPL [21]. This section also introduces the most common global constraints used in constraint models, and the running example we will use in this paper to illustrate our method.

2.1. The MiniZinc modelling language by means of our running example

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. Let us illustrate this by presenting a model for our running example: the *Progressive Party Problem*. The goal in solving this problem is to organise a party between the crews of several boats at a yacht club. The rules of the party state that only certain boats can host visitors, while the crews of the remaining boats must take turns visiting these host boats at different times. Further, the host boats have capacities that limit the number of guests they can accommodate, each guest crew can only visit a host once, and guest crews cannot meet more than once.

This problem was introduced by [22], which compared a zero-one integer formalisation – a specialisation of Mixed Integer Programming (MIP) involving only zero-one variables – against a Constraint Programming (CP) one. Typically, MIP formulations require many more constraints to describe a problem than CP ones. The MIP model suffered from this issue and, as a result, the great number of constraints prevented the solver from finding a solution, while the CP model was able to successfully find a 13-host solution.

A MiniZinc version of the CP model from [22] is shown in Listing 1. Lines 1–6 declare the parameters of the model. Line 1 declares the first three parameters: the number of time periods `p`, number of host boats `nh`, and number of guest crews `ng`. These three parameters are used in lines 2–4 to declare the sets of designated host boats `HostBoats`, guest crews `GuestCrews` and time periods `Time`, where each host boat, guest crew and time period is identified by a distinct integer in the ranges `1..nh`, `1..ng` and `1..p`, respectively. These sets are used in lines 5 and 6 to declare two arrays of parameters representing the number of members in each guest crew and the maximum capacity of each boat, respectively. The values for all these parameters are provided in data files, such as the following one:

```
% party.1.dzn
crew = [6, 5, 4, 4, 4, 3, 2, 2, 2, 2];
capacity = [7, 10, 8, 8, 11];
nh = 5;
ng = 10;
p = 3;
```

```
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)     % at_most_one_visit
16    /\ forall (t in Time)
17      (sum (g in GuestCrews)
18        (crew[g]*visits[g,h,t])<=capacity[h]));        % 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
25
26 solve satisfy;
27 output [show(hostedBy)];
```

Listing 1: MiniZinc model for the Progressive Party Problem based on the CP model of [22].

The *solution* variables in the model are declared in line 8 by a two dimensional array, where the variable selected by the array access `hostedBy[g,t]` represents the boat in `HostBoats` that hosts guest crew `g` at time `t`, with `g` coming from the set `GuestCrews`, and `t` coming from the set `Time`. Lines 9–11 declare a three dimensional array of auxiliary zero-one variables satisfying a constraint (labelled `channel`) that ensures auxiliary variable `visits[g,h,t]` is 1 if and only if `hostedBy[g,t]=h`. That is, it is 1 if guest crew `g` visits host boat `h` at time `t`. These auxiliary variables are used in lines 13–18 to (a) constrain each guest crew `g` to visit each host boat `h` at most once (by summing up the value of `visits[g,h,t]` for every time period `t`, and ensuring the sum is less than or equal to 1); and (b) ensure the capacity constraints are satisfied (by summing up the members `crew[g]` of any guest crew `g` that visits the same host boat `h` at the same time `t`, and ensuring

that the sum is less than or equal to `capacity[h]`). Note that constraint (a) is set by lines 13–15 (labelled `at_most_one_visit`), while constraint (b) is set by line 13 (labelled `capacity`) together with lines 16–18. This is achieved by using conjunction (represented by the symbol `/\`) and nested forall loops to build a *conjoined constraint*. This conjoined constraint could easily be replaced by two constraints, by repeating line 13 right before line 16 (and appropriately dealing with parentheses and semicolons). While this might be considered simpler, it might also be considered less structured and, thus, less clear to other users, as they would be required to notice that both loops use the same index sets.

Finally, lines 20–24 introduce another three dimensional array of auxiliary zero-one variables with a constraint (labelled `will_meet`) that ensures that the auxiliary variable `meet[k,l,t]` is 1 if and only if `hostedBy[k,t]=hostedBy[l,t]`. That is, it is 1 if guest crews `k` and `l` meet at time period `t`. These auxiliary variables are then used to constrain each pair of guest crews to meet at most once (labelled `meet_once`). Again, the constraints are conjoined and nested within the outer forall set in line 20.

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: $(\forall GHT : \mathit{channel}) \wedge (\forall H : (\forall G : \mathit{at_most_one_visit}) \wedge (\forall T : \mathit{capacity})) \wedge (\forall GG : (\forall T : \mathit{will_meet}) \wedge \mathit{meet_once})$, where *channel* denotes the constraint appearing in lines 10–11, *at_most_one_visit* that in line 15, *capacity* that in line 18, *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 index sets, e.g., $\forall GHT$ is equivalent to $\forall T \forall GH$, to $\forall H \forall T \forall G$, and so on.

As mentioned before, the method proposed in this paper can be adapted for any other constraint modelling language that supports global constraints. This is emphasised not only by the above shorthand notation for loops, but also by the mathematical notation we have used throughout all our algorithms.

2.2. Common Global Constraints

The Global Constraint Catalog maintains a large catalogue of definitions and implementations of global constraints, as well as their availability in different solvers [23]. As mentioned in Section 1, one of the most common global constraints is `alldifferent`(xs), which is equivalent to (i.e., can be *decomposed* into) the following set of binary constraints $\forall v_1 \in xs, v_2 \in xs \setminus \{v_1\} : v_1 \neq v_2$. Note

that the number of variables in *xs* might only be known once the model is instantiated with input data.

A common class of global constraints is that of *channelling constraints*, which establish a bijective connection between two different kinds of variables that represent the same information. This can be useful either because some constraints are easier to express when using a given type of variable, or because the aim is to create two alternative models (often referred to as duals) that can benefit from each other's solving by means of the channelling constraints. There are two main channelling constraints. The first is `inverse`(xs, ys), which connects the two arrays of integer variables *xs* and *ys* as follows: $\forall i, j : xs[i] = j \leftrightarrow ys[j] = i$. This constraint is often used when modelling problems involving successor and predecessor variables, with $xs[i]$ indicating the task that will follow task i, and $ys[j]$ indicating the task that should precede task j. The second one is `channel`(x, zs), which connects an integer variable x with an array of zero-one variables *zs* as follows: $x = j \leftrightarrow zs[j] = 1$. This is useful for constraints that require a binary variable *view* of some integer variable, or vice-versa.

Another common global constraint is `bin-packing-capa`(cs, bs, ws), where objects in array *bs* with weights given by array *ws* are packed in bins, such that the capacities of the bins given by array *cs* are not violated. Formally, the global constraint can be decomposed as $\forall j : cs[j] \geq \sum_i ws[i](bs[i] = j)$, where parameter $ws[i]$ gives the weight of object i, parameter $cs[j]$ provides the capacity of bin j, and variable $bs[i] = j$ iff object i is packed in bin j.

An expert modeller can easily see that line 15 in Listing 1 represents an `alldifferent` constraint on the `hostedBy` variables for each `g in GuestCrews`, that is, `alldifferent([hostedBy[g,t] | t in Time])`. This can be seen by examining the `channel` constraint that connects `visits` to the `hostedBy` variables. Since for a fixed guest and host only one `visits` variable will be set to 1, a host can only be assigned once for a fixed guest and time in the `hostedBy` variables. It is less obvious that line 18 can be expressed with the `bin-packing-capa` constraint with arguments: `capacity`, `[hostedBy[g,t] | g in GuestCrews]`, and `crew`. The *capacity* constraints encode this with the relationship $x_i = j$ being provided by the `channel` constraints. Our goal is to automatically detect these global constraints so that a user may add them to their model. Note that this is difficult because the resulting global constraint is *parametric*, since one of the arguments is coming from the input data (see Section 4.4.1).

3. 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, the closest works are in [19, 18], which introduced Constraint Seeker to infer global constraints from positive and negative examples of solutions, and Model Seeker to infer an entire instance, given a set of complete solutions to a constraint problem. Constraint Seeker and Model Seeker differ from the method presented here both in motivation and methodology. The motivation of our paper is to identify parts of a given model that can be replaced by global constraints. Having access to an initial model is key for our methodology, as it allows us to make extensive use of the explicit 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 auxiliary variables used in the model and its input data. Having access to the input data is also key for our methodology, as it allows us to (a) better generate candidates and (b) automatically generate as many solutions as required for ranking the candidate global constraints. For example, as shown in Section 6, the input data enables us to derive `bin-packing-capa` constraints for the Progressive Party problem. In contrast, Model Seeker cannot infer these from the solutions alone, since the constraints depend on fixed parameters for the weights and capacities that are only available in the input data.

The method presented here also relates to the CGRASS system [12, 13], which among other transformations, syntactically matches cliques of dis-equality constraints that can be replaced by an `alldifferent` constraint. This requires the modeller to implement the constraint using the exact syntactic structure to be matched. The approach taken in this paper goes well beyond this. First, our approach is not restricted to syntactic matching, but infers global constraints based on their semantics (approximated by a sample of their solution space). As a result, our approach is not limited to a hard-coded set of global constraints, it can infer any global constraint supported by the modelling language. And second, our approach infers global constraints that can be added to the model and, thus, can benefit all its instances, rather than the single instance being considered. Syntactic matching could be used, however, to detect common formulations.

Other acquisition approaches focus on the automatic generation of implied constraints. One of the most general methods, described in [17], uses machine

learning to induce constraints from the solutions of small problems, and a theorem prover to show that these constraints hold for the model. Its generality comes at a cost, since its application is limited to a small class of problems: the model can only be parameterised by a single integer, and it needs to be expressible in first order logic. In our case the constraints are already predetermined (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 in general is undecidable. Rather, our method *suggests* candidate constraints to the user.

Another related method is that of CONACQ [24, 25] 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. It uses the 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 this method is general and powerful, it considers instances, rather than models themselves. Further, the library of constraints must be hand tailored for a particular problem since for each constraint in the library, many feasible instantiations must be enumerated. For example, to acquire the constraints of a Sudoku puzzle, with 81 variables, each binary constraint in the library will add up to 81^2 constraints to be considered in the version space. In contrast, our method can infer model constraints and can handle much larger libraries.

Finally, [16] describes how implied parametric constraints can be learned by first adding a large disjunction of constraints with different parameters that together are trivially implied, and then successively pruning that disjunction by removing constraints that do not change the set of solutions found. This method goes further than ours, as it infers parameters from solutions, while we only consider parameters present in the model. However, the method is only applicable for constraints for which a trivially true disjunction can be constructed efficiently, i.e., where the size of that disjunction does not become prohibitively large. In those cases, the method could be used in conjunction with ours. For functional dependencies, we can infer parameter sets, as in the Schedule example in Section 7.

In the area of *model transformation*, the work on ESSENCE [1, 26] and ESSENCE' [14, 15, 27] is somewhat related. These works transform an ESSENCE model, often specified in a highly abstract manner, into a lower-level one specified in ESSENCE'. In such cases, there is no need to detect any "implicit model structures" as such structures would already be present in the ESSENCE specification. Our method moves in the opposite direction: we detect parts of a model that can be expressed

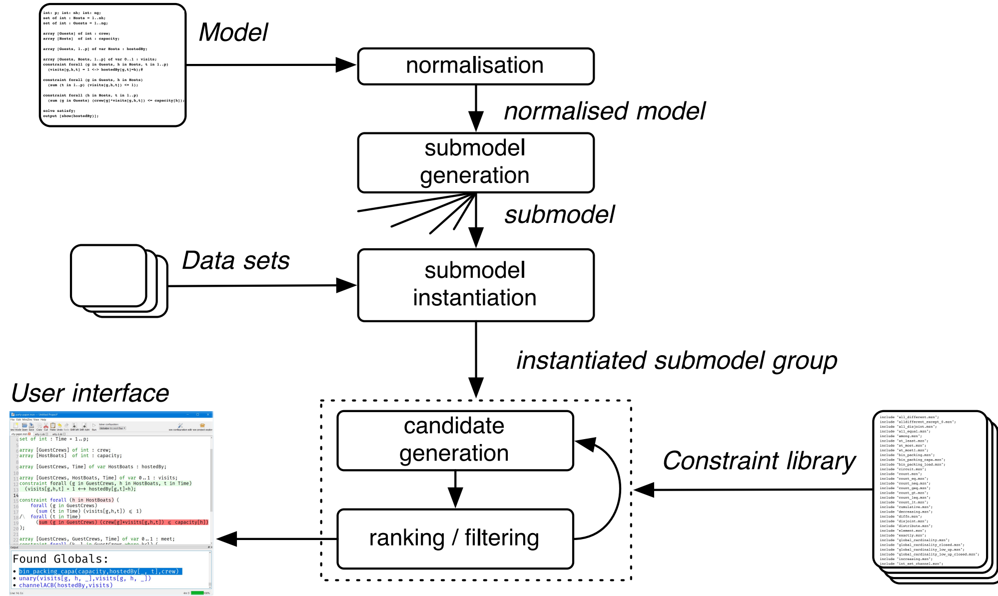


Figure 1: Graphical overview of model globalization.

with a more generic (i.e., higher-level) pattern. Note that, while we currently focus on global constraints for our generic patterns, it is straightforward to extend the method to use any other useful constraint pattern (for example, identifying representations of sets/multi-sets, that ESSENCE can exploit, in a manner similar to how channelling constraints are found in Section 5.2). In fact, globalization and automatic transformation are complementary: starting from a low-level model, globalization yields a higher-level model that might then be amenable to more sophisticated automatic transformations. Further, users of ESSENCE might model the problem using the lower-level features of ESSENCE', in which case our method can be useful for ESSENCE too.

Algorithm 1 High-level algorithm for Globalizer.

procedure `globalize`(M, $DtFiles$, Lib, $SubModelSize$)
  $\langle Constrs, Decls \rangle \leftarrow$ `normalize`(M)
  $Submodels \leftarrow$ `generate_submodels`($\langle Constrs, Decls \rangle$, $SubModelSize$)
  $Candidates \leftarrow \emptyset$
  for each $Submodel \in Submodels$ **do**
    $SubmodelInstanceGroup \leftarrow$ `instantiate_submodel`($Submodel$, $DtFiles$)
    $Candidates \leftarrow Candidates \cup$ `process_group`($SubmodelInstanceGroup$, Lib)
  return $Candidates$

4. Globalization

Figure 1 provides a graphical overview of the main steps of the method, which is also presented in high-level algorithmic form in Algorithm 1. Intuitively, the method starts by *normalizing* the input model M, that is, by separating the declarations *Decls* from the constraints *Constrs*, and normalizing the constraints in such a way that conjoined constraints are split into their component constraints. After normalization, several *Submodels* are generated as targets for globalization, where each such target *Submodel* $\in$ *Submodels* corresponds to the submodel formed by combining *Decls* with a particular subset of the normalized constraints in *Constrs*. The size of the submodel is controlled by the input parameter *SubModelSize*, which indicates the maximum number of constraints that will appear in the submodel. Each such submodel is then instantiated with the data provided by each data file appearing in the set *DtFiles*, obtaining a group of submodel instances. Each such group is then processed to determine its associated set of *candidate global constraints*.

A candidate global constraint needs to have two properties. First, it needs to be *implied by* each submodel instance in the group. This is important because if the candidate was not implied it could remove solutions of the original problem – it would therefore be incorrect to add it to the model. Secondly, the candidate should also *imply* each submodel instance. If it does, it is in fact equivalent to the submodel, and we can replace the submodel by the candidate. If it does not imply an instance, it means that the candidate is *weaker* than the submodel, but it may still be a valid and useful implied constraint. Determining whether a constraint is implied by or implies a model is not trivial. Instead of attempting to prove these two properties, our method is based on *sampling*. If, for each submodel instance in a group, a sufficient number of randomly sampled solutions of that submodel instance satisfy a constraint, it is highly likely that the constraint is implied by the submodel. Conversely, if a sufficient number of randomly sampled solutions of a candidate constraint satisfy all submodel instances, it is highly likely that the candidate implies the submodel.

Therefore, the processing of each group of submodel instances first generates an initial set of candidates from those present in the constraint library *Lib*, and then uses sampling to *score* them according to how well their solution spaces match that of the submodel instance group, and *filter them out* if their score is below a given threshold. Note that the instantiation of the submodel is needed to be able to sample its solution space and test its equivalence with that of the global constraint. Finally, the union of all filtered candidates is returned and, as shown

in Section 6.3, the set is presented to the user by means of an interactive GUI. The following subsections discuss each of these steps in detail.

4.1. Normalization

The algorithm for normalizing a model M is very simple and is shown in Algorithm 2. The procedure partitions M into two sets: the set *Constrs* of *normalized* constraints and the set *Decls* of original variable and parameter declarations. The constraints in the model are normalized by exhaustively applying two rewriting rules that (a) split conjunctions of the form $c_1 \wedge \ldots \wedge c_n$, into their individual constraints $c_1, \ldots, c_n$, and (b) split `forall` loops that contain conjunctions into individual `forall` loops. For example, in the case of the Progressive Party model, *Decls* is initialised as the set containing all variable declarations in the model, that is, all declarations appearing in lines 1–9 plus the one in line 20. *Constrs* is initialised as the set containing the three top-level constraints in the model, that is, $\{\forall GHT : \mathit{channel}, \forall H : (\forall G : \mathit{at_most_one_visit}) \wedge (\forall T : \mathit{capacity}), \forall GG : (\forall T : \mathit{will_meet}) \wedge \mathit{meet_once}\}$. After normalizing the Progressive Party model, *Constrs* contains the following constraints $\{\forall GHT : \mathit{channel}, \forall HG : \mathit{at_most_one_visit}, \forall HT : \mathit{capacity}, \forall GGT : \mathit{will_meet}, \forall GG : \mathit{meet_once}\}$.

Normalization is vital for discovering global constraints that describe parts of a constraint, rather than all of it. For example, in the Progressive Party the constraint $\forall HG : \mathit{at_most_one_visit}$, combined with the channelling constraint $\forall GHT : \mathit{channel}$, is equivalent to a conjunction of several `alldifferent` constraints. To discover this, each component constraint must be considered separately.

There are other normalization steps that could be implemented. For example, the branches of an `if` statement occurring in a loop can be separated into individual top level constraints that isolate each branch, or a **`let`** expression could

---

Algorithm 2 Flatten conjunctions and aggregate `forall` quantifiers to normalize.

---

procedure `normalize`(M)
    Decls $\leftarrow$ set of all variable and parameter declarations in M
    Constrs $\leftarrow$ set of constraints in M
    while one of the following rules applies **do**
        if there is a $c \in \mathit{Constrs}$ of the form $(c_1 \wedge \ldots \wedge c_n)$ **then**
            $\mathit{Constrs} \leftarrow (\mathit{Constrs} \setminus c) \cup \{c_1, \ldots, c_n\}$
        if there is a $c \in \mathit{Constrs}$ of the form $(\forall A_1 \ldots \forall A_n : c_1 \wedge \ldots \wedge c_m)$ **then**
            $\mathit{Constrs} \leftarrow (\mathit{Constrs} \setminus c) \cup \{\forall A_1 \ldots \forall A_n : c_1, \ldots, \forall A_1 \ldots \forall A_n : c_m\}$
    return $\langle \mathit{Constrs}, \mathit{Decls} \rangle$

---

be hoisted to allow its constraints to be accessed individually. We limit our normalization to the simple rules described in Algorithm 2, as they cover the most common loop types that occur in the MiniZinc benchmarks and are enough to demonstrate the usefulness of our method: they allow us to fully unroll and normalise more than 50% of the loops used in the benchmark suite. The handling of `if` expressions around loops or `if` expressions inside loops, would only allow us to cover a further 10%. Note that while adding more rules may increase the accuracy of the system, it will also increase the number of submodels to be explored significantly.

4.2. Generating Submodels

After normalizing model M, the next step is to generate the submodels of M that will be considered as targets for globalization. The goal is to find as many combinations of M's constraints as possible, since we may be able to find an equivalent global constraint for them. Algorithm 3 shows how this is achieved by the `generate_submodels` procedure, which takes as arguments the normalized

Algorithm 3 Splitting a set *Constrs* of constraints into submodels.

procedure `generate_submodels`($\langle Constrs, Decls\rangle$, *SubModelSize*)
    $targetSet \leftarrow \emptyset$
    for each ($S \subseteq Constrs$ **where** $1 \leq |S| \leq SubModelSize$) **do**
        $Decls' \leftarrow$ subset of *Decls* that includes variables and parameters of S
        $targetSet \leftarrow targetSet \cup \{\langle S, Decls'\rangle\} \cup$ `unrollings`($\langle S, Decls'\rangle$)
    return *targetSet*
procedure `unrollings`($\langle Constrs, Decls\rangle$)
    $submodels \leftarrow \emptyset$
    if *Constrs* are of the form $(\forall a_1 \in A_1, \forall a_2 \in A_2, \ldots \forall a_n \in A_n \langle \texttt{where } c\rangle) : e)$ **then**
        for each $(X \subseteq \{a_1 \in A_1, a_2 \in A_2, \ldots a_n \in A_n\})$ **do**
            if X is not empty **then**
                if *Constrs* has $\langle \texttt{where } c\rangle$ that depends on X **then**
                    $Constrs' \leftarrow (c\text{->}Constrs)$ with X removed from the forall
                else
                    $Constrs' \leftarrow Constrs$ with X removed from the forall
                for each ($a_i \in A_i$ **in** X) **do**
                    $Constrs' \leftarrow Constrs' \cup \{a_i = val(A_i)\}$
                $submodels \leftarrow submodels \cup \{\langle Constrs', Decls\rangle\}$
    return *submodels*

```
int: p; int: nh; int: ng;
set of int: HostBoats = 1..nh;
set of int: GuestCrews = 1..ng;
set of int: Time = 1..p;
array [GuestCrews, Time] of var HostBoats : hostedBy;
array [GuestCrews, HostBoats, Time] of var 0..1: visits;
constraint forall(g in GuestCrews, h in HostBoats, t in Time)
    ( visits[g,h,t] = 1 <-> hostedBy[g,t]=h );
```

Listing 2: A submodel generated for the *Progressive Party Problem* model in Listing 1

set of constraints (*Constrs*), the declarations (*Decls*), and the maximum number of constraints to include in each submodel (*SubModelSize*), and generates all submodels that can be formed by conjunctions of at most the given number of constraints (our implementation uses *SubModelSize=2* as default, as this shows the best trade-off between accuracy and runtime; see Section 7.2).

The procedure works as follows. For each subset $S \in Constrs$ of appropriate size, it computes the subset $Decls'$ of *Decls* of variables and parameters in S, and adds the submodel $\langle S, Decls' \rangle$ to the set *targetSet*. For example, Listing 2 shows a generated submodel that corresponds to the $\forall GHT : channel$ constraint from the model in Listing 1. Note that the declarations for the `crew` and `capacity` arrays (declared in the model) are not added to $Decls'$, as these arrays do not appear in the constraints of the submodel. As we will see later, using $Decls'$ rather than *Decls* can significantly improve the efficiency of our method.

The procedure can also add *unrollings* of submodel $\langle \mathrm{S}, Decls' \rangle$ to *targetSet*. Unrollings are used to strengthen the global constraints we find by forcing corner cases (such as the first and last iteration of a comprehension) to be handled. A submodel $\langle Constrs, Decls \rangle$ will be unrolled only if *Constrs* has constraints of the form $(\forall a_1 \in A_1, \forall a_2 \in A_2, \ldots \forall a_n \in A_n \langle \texttt{where } c \rangle) : e)$, that is, they are defined over a (possibly nested) universally quantified expression, which might or might not have a `where` condition c. If so, unrolling is achieved by selecting some subset X of the quantified variables and constructing a new set of constraints $Constrs'$ where each of these variables $a_i \in X$ is removed from the quantified part and equated to expression $val(A_i)$, denoting a value in domain set A_i. This is achieved by adding the relevant $a_i = val(A_i)$ constraints to each $Constrs'$. Later, when the submodel is instantiated, these values will be fixed to an arbitrary value of their domain. If the condition c depends on the quantified variables in X, c is moved from the `where` part into the constraint e as an implication. Listing 3 shows a partial unrolling of

```
int: g = val(GuestCrews);
constraint forall (h in HostBoats, t in Time)
    ( visits[g,h,t] = 1 <-> hostedBy[g,t]=h );
```

Listing 3: Partial unrolling of the constraint in Listing 2.

```
% Constraints in a generated submodel, which we call SubM
constraint forall(h in HostBoats, t in Time)
    (sum(g in GuestCrews)(crew[g]*visits[g, h, t])<=capacity[h]);
constraint forall(k,l in GuestCrews, t in Time where k < l)
    (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% SubM1: unrolling of SubM with t fixed & removed from the 2 foralls
t = val(Times);
constraint forall(h in HostBoats)
    (sum(g in GuestCrews)(crew[g]*visits[g, h, t])<=capacity[h]);
constraint forall(k,l in GuestCrews where k < l)
    (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% Constraint in a generated submodel, which we call SubM'
constraint forall(k,l in GuestCrews where k < l)
    (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% SubM'1: unrolling of SubM' with l fixed and removed
l = val(GuestCrews)
constraint forall(k in GuestCrews where k < l)
    (sum(t in Time)(meet[k, l, t]) <= 1);

% SubM'2: unrolling of SubM' with k fixed and removed
k = val(GuestCrews);
constraint forall(l in GuestCrews where k < l)
    (sum(t in Time)(meet[k, l, t]) <= 1);

% SubM'3: unrolling of SubM' with k and l fixed and removed. Note
% that the condition from the forall now implies the constraint.
k = val(GuestCrews); l = val(GuestCrews);
constraint k < l -> sum(t in Time)(meet[k, l, t]) <= 1;
```

Listing 4: Constraints in two submodels, *SubM* and $SubM'$, and some of their loops unrolled.

the constraint in the submodel of Listing 2.

To illustrate how the instantiation of $val(A_i)$ works, Listing 4 shows two more submodels *SubM* and $SubM'$ generated by our method for the model M in Listing 1, together with one and three submodels, respectively, obtained by the partial

Algorithm 4 Instantiating a set of submodels.

```
procedure instantiate_submodel(⟨Constrs, Decls⟩, DtFiles)
    for each DtFile ∈ DtFiles do
        Decls ← Decls ∪ DtFile                          ▷ Assign parameters with DtFile
        TmpSetDecls ← {Decls}
        for each declaration of the form x = val(A_x) ∈ Decls do
            for each TmpDecls ∈ TmpSetDecls do
                TmpSetDecls ← TmpSetDecls \ TmpDecls
                for each value v ∈ {min(A_x), max(A_x), mid(A_x)} do
                    TmpSetDecls ← TmpSetDecls ∪ (TmpDecls \ x = val(A_x) ∪ {x = v})
    SubmodelInstanceGroup ← ∅
    for each TmpDecls ∈ TmpSetDecls do
        if ∀c1, c2 ∈ Constrs, c1 and c2 are connected then
            SubmodelInstanceGroup ← SubmodelInstanceGroup ∪ {⟨Constrs, TmpDecls⟩}
    return SubmodelInstanceGroup
```

unrollings of their constraints. For brevity, the declarations *Decls* in all submodels have been omitted, but they contain all declarations in M for the first two submodels (*SubM* and *SubM*1), and lines 1 to 6, 8, and 20 in M for the last four submodels ($SubM'$, $SubM'1$, $SubM'2$ and $SubM'3$). Submodel *SubM* has two constraints, which are being considered as candidate targets for globalization by the algorithm. Submodel *SubM*1 is obtained by unrolling the constraints in *SubM*, where the value for `t` has been fixed, and `t in Time` has been removed from the `forall` loops of both constraints. Submodel $SubM'$ has a single constraint. $SubM'1$ and $SubM'2$ unroll it by fixing variables `l` and `k`, respectively, and removing them from their respective loops. The final submodel also unrolls the constraint in $SubM'$ and shows what happens when all loop variables are fixed and there is a `where` condition (`k < l`) that must be considered: the condition implies the remaining constraint. In this way, the implication (`k < l -> ...`) ensures that candidates cannot be compared against this submodel when the condition is not met.

4.3. Instantiating Submodels

Each submodel generated in the previous step will give rise to a group of concrete instances, constructed by instantiating the unrolled variables with concrete values and by adding given data values for the model's parameters.

This is achieved by the procedure presented in Algorithm 4, which takes as input a submodel $\langle Constrs, Decls \rangle$ and a set of data files *DtFiles*, and proceeds as

follows. It first constructs in *TmpSetDecls* all possible instantiations of the declarations using each of the different input data files together with some of the values of the unrolled variables. As discussed later in Section 6.1, while we have used $min(x)$, $max(x)$, and $mid(x)$ for each unrolled variable x, representing the minimum, maximum and mid value, respectively, of its domain, any other concrete values could have been used. Listing 5 shows the constraints in the submodel of Listing 2 after being partially unrolled and instantiated with `g` being fixed to the minimum value from the set `GuestCrews`. Algorithm 4 proceeds after computing *TmpSetDecls* by combining each of its elements with the constraints *Constrs* in the submodel to produce a submodel instance. Note that this last step only happens if all pairs of constraints in the resulting instance are *connected*, that is, if the constraints either directly share at least one uninstantiated variable, or there is a path of connecting constraints for them in *Constrs*, such that every two adjacent constraints in the path are connected. This is a reasonable optimisation as constraints represent a relationship between variables and, if there is no relationship in the model, one cannot possibly be inferred.

4.4. Processing Submodel Instance Groups

Once procedure `instantiate_submodel`(*Submodel*, *DtFiles*) finishes, it returns in *SubmodelInstanceGroup* a set of instances of *Submodel* constructed using *DtFiles*. Recall that each such instance has different parameter values either due to different data files given by the user, or because different values were chosen (from $min(x)$, $max(x)$, and $mid(x)$) during loop unrolling. For each *Submodel* of the original model M, its associated *SubmodelInstanceGroup* is then processed in order to generate, rank and filter a set of candidate constraints, which are then added to the final set *Candidates* for M. This is achieved by the procedure `process_group`(*SubmodelInstanceGroup*, *Decls*, *Lib*) (shown in Algorithm 5), where *SubmodelInstanceGroup* is as before, *Decls* is the set of declarations in the original model, and *Lib* is the global constraint library.

Processing all instances of each submodel as a group allows `process_group` to increase its accuracy, by taking the intersection of the candidate constraints

```
int: g = min(GuestCrews);
constraint forall (h in HostBoats, t in Time)
    ( visits[g,h,t] = 1 <-> hostedBy[g,t] = h );
```

Listing 5: Unrolled constraint from the submodel in Listing 2.

Algorithm 5 Processing submodel instance groups.

```
procedure process_group(SubmodelInstanceGroup, Decls, Lib)
    Candidates ← Universe
    for each SubModelInstance in SubmodelInstanceGroup do
        Candidates ← Candidates ∩ generate_candidates(SubModelInstance,
Decls, Lib)
        Candidates ← rank_and_filter(SubModelInstance, Candidates)
    return Candidates
```

found for each *SubModelInstance* in *SubmodelInstanceGroup*. This is correct because a global constraint can only be equivalent to a submodel (and, thus, be a candidate) if it holds for every instantiation of the submodel. Initially, the full set of constraints and argument tuples (written as the special symbol *Universe*) is considered as the set of possible candidates. Thanks to the filtering (intersection) performed later in the process, the number of candidates in this initial set may decrease with each subsequent instance in the group, and only those that remain after processing the final instance are returned as candidates. These remaining candidates are exactly the intersection we seek to compute.

4.4.1. Candidate Generation

The first step of the group processing is the generation of candidate constraints for each *SubmodelInstance* in *SubmodelInstanceGroup*. The algorithm for generating candidates is shown in Algorithm 6. It takes as input a *SubmodelInstance* in its associated *SubmodelInstanceGroup*, the set *Decls* of declarations of the original model *M*, and the library of global constraints *Lib*. Note that each constraint entry in *Lib* has a signature comprising the name, arity, and valid argument types. 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 6.2 for details on the particular *Lib* used by our implementation.

The algorithm proceeds as follows. First, it finds a reasonably large random sample of solutions of the given *SubModelInstance* (the default number of samples is 30, see Section 7 for details), called *Solutions*. Intuitively, we will consider a global constraint as a candidate for the submodel if it is satisfied by *all* these sample *Solutions*.

Possible candidate constraints are obtained by combining each global constraint cons in *Lib* with an A-tuple args of arguments, where A is the arity of cons. The arguments are drawn from the identifiers that appear in *SubModelInstance* as

Algorithm 6 Generating candidate constraints for a submodel instance.

```
procedure generate_candidates(SubModelInstance, Decls, Lib)
    Candidates ← ∅
    Solutions ← set of random sample of solutions of SubModelInstance
    Template ← (SubModelInstance \ Decls)
    BaseArguments ←
            (variable and parameter collections in SubModelInstance) ∪
            (variable and parameter sub-collections in constraints of SubModelInstance)
    Arguments ← BaseArguments ∪
            (array accesses of elements of BaseArguments) ∪
            { constant 0 } ∪ { blank symbol }
    for each constraint cons in Lib do
        for each tuple args that can be built from Arguments do
            Replace blank symbols in args by their value if necessary
            implied ← true
            for each s ∈ Solutions do
                Instance ← Template ∪ s ∪ (constraints for cons(args))
                if Instance is unsatisfiable then
                    implied ← false
                    break
            if implied then
                Candidates ← Candidates ∪ {cons(args)}
    return Candidates
```

shown in Algorithm 6. The *BaseArguments* include the variable and parameter collections whose identifiers are referenced in *SubModelInstance*, plus the same collections restricted to the subsets that are actually used in the constraints of *SubModelInstance*. In addition, the arguments can also be array access expressions composed from the *BaseArguments*, 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. For example, in the constraint `maximum(_, xs)` the value of the first argument can be computed from the value of the second and this will be represented by a blank symbol. Note that the value of the blank symbol 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 — for example, if one sample solution suggests `maximum(10,xs)` but another says `maximum(8,xs)` — the candidates will be discarded in the intersection

computation in `process_group`. Note also that, since the generated candidate constraint must be the same for all instances, the functionally-defined arguments to this constraint must take the same value *across all instances* to be considered as a candidate, since the constraint cannot be posted to the model without a selected value. This is however not a significant issue, as they are only used when no named parameter is found (i.e., `maximum(a, xs)`, where `a` comes from the model, should be taken over `maximum(8, xs)`).

The combination of constraint and arguments is then tested against all solutions $s \in \mathit{Solutions}$. If any s does not satisfy $\mathsf{cons}(\mathsf{args})$, then this cannot be an implied constraint (as it would remove valid solutions of the submodel), and it will not be added to the set of candidates. If all solutions satisfy the constraint, it is recorded as a candidate for *SubModelInstance*.

Let us show how `generate_candidates` works with the *SubModelInstance* 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 $\mathit{DtFile} \in \mathit{DtFiles}$. Listing 5 presents a MiniZinc instance that corresponds to the selected submodel instance using the data presented in Section 2. The *BaseArguments* computed by the procedure for *SubModelInstance* 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* computed are the constant 0, the blank symbol, the base arguments, and the array accesses formed by combining an array with a parameter, e.g., `crew[nh]` and `hostedBy[p, ng]`. After considering all constraints in *Lib* with these arguments, the procedure generates 39 candidate constraints including the following three:

```
lex2(hostedBy)
alldifferent([hostedBy[g,1], hostedBy[g,2], ..., hostedBy[g,p]])
sub_circuit([hostedBy[g,1], hostedBy[g,2], ..., hostedBy[g,p]])
```

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

4.4.2. Ranking and Filtering

As shown in Algorithm 5, once all candidates for a submodel are generated, the method calls procedure `rank_and_filter` to rank and filter those candidates. Recall that all candidate constraints are assumed to be implied by the sub-

model (with sufficient probability, depending on the number of samples used). The ranking step decides how likely it is for the candidate to imply the submodel.

This procedure, defined in Algorithm 7, works as follows. For each candidate constraint `cons`(args) inferred for a given *SubModelInstance* (and, given the intersection performed by `process_group`, for all previously computed and remaining candidates too), `rank_and_filter` measures how closely `cons`(args) matches *SubModelInstance*. To achieve this, it collects a random sample of solutions of `cons`(args), and computes the fraction of these that are also solutions to *SubModelInstance*. If the constraint is equivalent to the submodel of *SubModelInstance*, this fraction must be 1; if the constraint is a poor match, the fraction is likely to be close to 0. The procedure then filters the candidates by removing those whose matching fraction is smaller than a given threshold. In our experience, a threshold of 0.5 has been sufficient to eliminate imperfect matches and to present interesting implied constraints for our wide range of benchmarks. However, since we are primarily concerned with replacements (i.e., equivalences), we have used a threshold of 1.0 in our experimental evaluation. A ranking of the candidates is achieved in `rank_and_filter` by simply sorting them by their *Score*.

Algorithm 7 Ranking submodel instances.

```
procedure rank_and_filter(SubModelInstance, Candidates)
    result ← ∅
    for each cons(args) in Candidates do
        SolutionsC ← random sample of solutions of cons(args)
        SolutionsM ← subset of SolutionsC that are also solutions of SubModelInstance
        Score ← |SolutionsM| ÷ |SolutionsC|
        if Score ≥ equivalenceThreshold then
            result ← result ∪ ⟨Score, cons(args)⟩
    return sequence of cons(args) from result ranked by Score
```

In some cases, constraints in a model are equivalent to a candidate global constraint only in the context of another constraint. Consider a *SubModelInstance* 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 *SubModelInstance* have the constraints $(t = \min(T)) \wedge \forall HG : \mathit{channel} \wedge \forall H : \mathit{capacity}$ from the Progressive Party model. 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*. This is due to the fact that the `bin-packing` constraint does not constrain the auxiliary `visits` variables. This connection must be provided by the `channel` constraints. In general, a contextually-equivalent constraint `cons`(args) will be implied by *SubModelInstance* but appear weaker than the instance and, thus, will be ranked lower. In this case, we use a constraint from *SubModelInstance* 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 (*SubModelInstance* $\setminus B$) 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 is thus considered to be a pair of the `cons`(args) and its context, which 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.

5. Preprocessing Steps

To improve effectiveness and performance, our method performs an initial pass looking for two particular kinds of generated submodel instance groups over a reduced library of global constraints. The first kind represents very weak groups of constraints that should be ignored in the main Globalizer pass. The second kind represents dual viewpoints that might help Globalizer uncover additional global constraints in the later pass.

5.1. Detecting Weak Groups

Globalizer often creates submodel instance groups that do not sufficiently constrain the variables of the problem. These groups will accept most sampled solutions of any candidate global constraint, but few of their sampled solutions would be accepted by these global constraints. Such weak groups are unlikely to result in any meaningful constraints and can cost Globalizer a lot of time. Fortunately, the detection of these groups is quite simple. Globalizer achieves this in its preprocessing pass by including the constant "`true`" as one of the global constraints in the reduced library. By doing this, for any given submodel instance group, Globalizer will generate a sample of solutions to its decision variables (which obviously satisfy global constraint `true`). If all these solutions are accepted by the group, it is removed. In addition, our implementation also detects (and removes) any groups that are a subset of an already detected weak group, to speed up processing.

```
predicate channel_2of3(array[int,int] of var int : x,
                       array[int,int,int] of var int : b) =
      lb_array(b) in {0,1}
   /\ ub_array(b) in {0,1}
   /\ forall (i in index_set_1of3(b),
              v in index_set_2of3(b),
              k in index_set_3of3(b))
        (x[i,k] = v <-> (b[i,v,k]=1));
```

Listing 6: Channelling between matrices of integer and binary variables.

5.2. Detecting Dual Viewpoints

The method proposed in Section 4.4.1 to build the arguments of the candidate global constraints uses only variable and parameter identifiers already present in the original model. This is not enough to discover constraints that might include dual “hidden” variables, that is, dual viewpoints involving new variables that could be declared and connected to existing model variables via channelling constraints. For example, it is common to model combinatorial problems using a set of zero-one variables b_v to represent a single integer decision x, such that $b_v = 1$ if and only if x takes the value v. By extension, a vector of integer decisions x_i can be represented as a matrix of zero-one variables b_{iv}, such that $b_{iv} = 1 \leftrightarrow x_i = v$. Further, a two dimensional matrix of integer decisions x_{ij} can be represented as a three dimensional matrix of zero-one variables b_{ijv}, such that $b_{ijv} = 1 \leftrightarrow x_{ij} = v$. The dimensions used to represent the variables can vary. For example, instead of the third dimension of b representing the selected value, the second dimension might be used. That is, instead of b_{ijk} representing $x_{ij} = k$ it could be $b_{ijk} = 1 \leftrightarrow x_{ik} = j$.

This form of channelling can be added to the model using the constraint shown in Listing 6. Note that the `IofJ` (`2of3`) part of the name indicates that this constraint is for channelling dimension `I` of a `J`-dimensional array of zero-one variables to a `J-1` dimensional matrix of integer variables. The predicate first posts two constraints restricting the domain of the variables in `b` to be zero-one. This is achieved using the `lb_array` and `ub_array` functions, which return the lowest lower bound and largest upper bound, respectively, of variables in the multi-dimensional array. Next, a `forall` loop iterates over the index sets of the `J`-dimensional array `b`, by means of the builtin MiniZinc function `index_set_IofJ`.

This style of representation is particularly common when targeting a mixed-integer programming solver. However, users may often omit the integer variables in such models and only declare the zero-one variables, as their corresponding

integer meanings can be computed from solutions. Automatically discovering these integer meanings may enable the discovery of global constraints that were previously obscured.

To find such "hidden" variables (in particular, implied integer variables), during the initial pass of Globalizer, our reduced library contains new global constraints that capture the dual representations we are trying to detect. In particular, we consider three global constraints which take a 1-, 2- or 3-dimensional array of zero-one variables as an argument, and hold if a specific dimension potentially represents the values of an integer. For example, the following constraint holds if a 3-dimensional array of variables `b` might represent a 2-dimensional array of integers `x` such that `b[i,v,k]=1 <-> x[i,k]=v`.

```
predicate binaries_represent_int_2of3(
              array[int,int,int] of var int : b) =
   lb_array(b) in {0,1}
/\ ub_array(b) in {0,1}
/\ forall (i in index_set_1of3(b),
           k in index_set_3of3(b))
     (sum (v in index_set_2of3(b))(b[i,v,k]) = 1);
```

If such a constraint is found during the initial pass, the implied dual variables (i.e, the integer variables) and a channelling constraint linking them to the original zero-one variables are added to the model. Once this initial pass has finished, the second pass of Globalizer will analyse the augmented model to find constraints using the original and the introduced variables.

Consider, for example, a model for the Latin Square problem. An $n \times n$ Latin Square is an $n \times n$ matrix of integers where each row and column contain permutations of the numbers 1 to n. The MiniZinc model for the Latin Square problem presented in Listing 7 is a summarised version of the one included in the MiniZinc benchmark set, which uses zero-one variables and linear constraints.

```
int: n;
set of int: N = 1..n;
array [N, N, N] of var 0..1: b;
constraint forall (i,j in N)(sum (k in N)(b[i,j,k]) = 1);
constraint forall (i,k in N)(sum (j in N)(b[i,j,k]) = 1);
constraint forall (j,k in N)(sum (i in N)(b[i,j,k]) = 1);
```

Listing 7: Binary model for Latin Squares problem.

The initial pass of Globalizer finds that each dimension of `b` might represent inte-

ger variables. That is, it finds the following constraints:

```
binaries_represent_int_1of3(b)
binaries_represent_int_2of3(b)
binaries_represent_int_3of3(b)
```

Then, during the ranking and filtering, it behaves as if the following arrays of auxiliary variables and channelling constraints (and their definitions) appeared in the model:

```
array [index_set_2of3(b),index_set_3of3(b)]
                        of var index_set_1of3(b): b_1of3;
constraint channel_1of3(b_1of3, b);
array [index_set_1of3(b),index_set_3of3(b)]
                        of var index_set_2of3(b): b_2of3;
constraint channel_2of3(b_2of3, b);
array [index_set_1of3(b),index_set_2of3(b)]
                        of var index_set_3of3(b): b_3of3;
constraint channel_3of3(b_3of3, b);
```

where the different versions of the channelling constraint map to different dimensions. As a result, Globalizer can find in its second pass the `alldifferent` constraints that correspond to the sets of sum constraints in the original model.

As another example, consider the zero-one integer model of the Progressive Party problem given by [22]. A notable difference between this model and the one presented in Listing 1 is the absence of the `hostedBy` variables. The discovery of the `alldifferent` and `bin-packing` constraints depended upon these variables being present. With the introduction of the additional initial pass, Globalizer can now discover these integer variables from the zero-one integer model and, therefore, the global constraints implied by the model. To illustrate this, we have translated the zero-one integer program into MiniZinc (see Listing 8) and adapted it in such a way that the set of hosts is fixed in advance, as done in Section 6 of [22]. The variable names are also changed to match those in Listing 1.

The first pass of Globalizer can discover that the `visits[g,h,t]` variables represent integer variables `H[g,t]` such that if `visits[g,h,t]` is 1, then `H[g,t]` equals `h`. After introducing these variables, the second pass is able to discover that the first constraint is an `alldifferent` constraint, and the second is a `bin-packing` constraint.

```
1  int : p; int : ng; int : nh;
2  set of int : HostBoats = 1..nh;
3  set of int : GuestCrews = 1..ng;
4  set of int : Time = 1..p;
5
6  array [GuestCrews] of int : crew;
7  array [HostBoats] of int : capacity;
8
9  array [GuestCrews, HostBoats, Time]
10         of var 0..1 : visits;
11
12 % at_most_one_visit
13 constraint forall (g in GuestCrews, t in Time) (
14     (sum(h in HostBoats)(visits[g,h,t])) = 1 );
15
16 % capacity
17 constraint forall (h in HostBoats, t in Time) (
18     (sum(g in GuestCrews)(
19          crew[g]*visits[g,h,t])) <= capacity[h]);
20
21 % will_meet
22 constraint forall (h1,h2 in HostBoats,
23                    g1,g2 in GuestCrews,
24                    t1,t2 in Time
25     where h1 != h2 /\  g1 < g2  /\ t1 != t2) (
26         visits[g1,h1,t1] + visits[g2,h1,t1] +
27         visits[g1,h2,t2] + visits[g2,h2,t2] <= 3 );
28
29 % meet_once
30 constraint forall (h in HostBoats, g in GuestCrews) (
31     (sum(t in Time)(visits[g,h,t])) <= 1);
```

Listing 8: Zero-one integer model for the Progressive Party Problem.

6. Implementation

This section provides details regarding the implementation of the Globalizer system used in the experiments. In particular, it discusses several implementation choices taken while implementing the different algorithms, describes the particular composition of the global constraint library used, and provides a quick overview of how Globalizer can be used through the MiniZinc IDE.

6.1. Implementation Choices

As shown in Algorithm 1, the submodels computed by `generate_submodels` can be processed independently. Thus, the implementation can process them in parallel to increase efficiency.

As shown in Algorithm 6, `generate_candidates`(*SubModelInstance*, *Decls*, *Lib*) first solves *SubModelInstance* to find a random sample of its solutions. To do this, the default implementation uses the standard MiniZinc tool `mzn2fzn` to flatten *SubModelInstance*, and the Gecode constraint solver [28] version 6.1.0 to find 30 random solutions for it (while 1, 30 and 100 were tested in our experiments, 30 was found to have the best accuracy/efficiency ratio and is, thus, used in the default implementation). These are found with a search that selects values in random order and restarts from scratch whenever a solution is found. If the search does not finish within one second, *SubModelInstance* is discarded as it may be too expensive to find 30 random solutions for this *SubModelInstance*.

Later in the process, the algorithm needs to determine the satisfiability of the *Instance* after adding candidate global constraints to the *Template*. Since this template has no variables, the added constraints are simply checked for satisfiability, for which no search is required. Such checks are performed very often, and it was clear that the expense of repeatedly flattening the instance and calling a full constraint solver was crippling. To avoid this, a simple evaluator of variable-less MiniZinc instances was implemented. In practice, this optimization is crucial, as the number of evaluations is usually in the hundreds of thousands.

Additionally, choices must be made about the number of samples that should be sought for any global constraint (as opposed to samples for a submodel instance), and the value of *SubModelSize*, i.e., the maximum number of constraints that can appear in the submodels. After considerable experimentation, we settled upon a maximum of two constraints in each submodel, and generating 30 samples for ranking and filtering in the default implementation. The results of our experiments are explored and evaluated in Section 7.

Another implementation choice concerns the values used to fix loop variables. We chose $min(x)$, $max(x)$, and $mid(x)$, as special cases in loops tend to be at the boundaries (e.g., the first or last variable in an array often has different constraints on it than the others). The $min(x)$ and $max(x)$ cover these edge cases, with $mid(x)$ selecting a non-boundary case. However, other values could also be selected.

6.2. Library of Global Constraints (second pass)

Table 1 lists the global constraints appearing in the current implementation of *Lib*, which is used for candidate generation. *Lib* is a subset of all the global con-

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

Table 1: Library of global constraints supported by Globalizer.

straints defined in MiniZinc's standard library. It does not contain any constraints over set variables (which are not yet supported by our Globalizer implementation), and omits several constraints that are very unlikely to ever match due to their complex parameters, such as table, regular and MDD constraints. Furthermore, we added the following two constraints:

- `channel(x, a)`: channels integer variable `x` to array of zero-one variables `a`.
- `gcc(x, counts)`: a special case of `global_cardinality` where the map from indices to values is fixed to the identity map.

These two global constraints appear in some popular solvers and are, therefore, worth detecting even if they are special cases of more general constraints. Our implementation is able to evaluate most of the global constraints in *Lib* using the default decomposition provided the MiniZinc library. However, some decompositions introduce variables that cannot be handled by the simple satisfiability check evaluator mentioned in Section 6.1. Thus, stand-alone constraint checkers were implemented in these cases to evaluate these globals.

As mentioned in Section 4.4.1, each global constraint is annotated with conditions for its use, to prevent the constraint from being considered as a candidate when it is trivially true or otherwise useless. For example, the `alldifferent` global 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)` global constraint specifies that every `n`-length subsequence of `x` must sum to a value between `l` and `u`. When generating candidates, we ensure that $l < u$ and $1 < n <$ `length(x)`. These two conditions ensure that the parameters make sense, thus avoiding the generation of

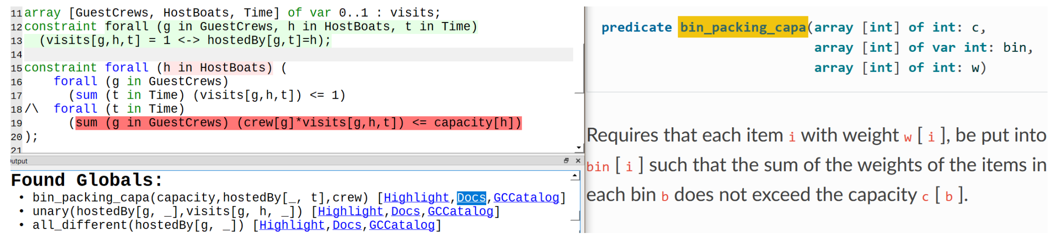


Figure 2: MiniZinc Globalizer interface showing highlighting and documentation.

meaningless candidates. Conditions on the parameters can be found in the global constraint catalog [23], where they are referred to as “restrictions”. These restrictions cannot always be directly translated for Globalizer as there are discrepancies between the signatures of MiniZinc constraints and the constraints in the catalog.

6.3. *The Globalizer tool*

Globalizer has been released as part of the binary MiniZinc distribution[1] and integrated with the MiniZinc IDE. The tool is implemented as a stand-alone “solver”, i.e., it can be run through the `minizinc` command line tool, as well as from the MiniZinc IDE. To use Globalizer with the IDE, a model and set of data files must be open in the IDE window. With Globalizer selected as the active solver, the user then should click the “Run” button and select the required data files. During execution, Globalizer updates a progress bar to provide feedback to the user. The current progress is based on the number of remaining submodel instance groups to be explored. Once complete, the tool will output the list of candidate global constraints that were found. For each suggested global constraint, the user can trigger the IDE to highlight the part of the original model that the candidate could replace, along with the candidate’s context, if any. The user is also presented with links to the MiniZinc documentation for a MiniZinc specific description of the constraint, and also to the Global Constraint Catalog [23] page for more detailed background information about this constraint and other related constraints.

Figure 2 shows a screen shot of the model for the progressive party problem open in the IDE. The user has executed Globalizer on this model with a pair of data files, and has selected the first candidate constraint returned by Globalizer. The highlighting indicates that it may be possible to replace the bounded

[1]See `https://www.minizinc.org/software.html` for download links and `https://www.minizinc.org/doc-latest/en/globalizer.html` for documentation.

`sum` constraint on the `visits` variables inside the top-level `forall` (highlighted in red), with a `bin_packing_capa` constraint on `hostedBy` variables in the context of the channelling constraint on lines 12-13 (highlighted in green). Links to documentation are also available for the candidate constraint. The official MiniZinc documentation for the selected constraint is shown on the right.

7. Experimental Evaluation

This section reports on two sets of experiments. The first one is designed to evaluate the accuracy and practicality of the default configuration of our implementation of Globalizer. The second set of experiments is designed to show the reasons behind the choices made for our default implementation.

7.1. Evaluating the Default Configuration

Let us start by discussing the results of evaluating the accuracy and practicality of the default configuration of the implementation (detailed in Section 6.1). The results are shown in Table 2 where, for each problem model, the table displays the name of the problem (**problem**), the time in seconds to run Globalizer (**time**), the number of global constraint candidates found with a score of 1 (|**cs**|), the number of submodel instance groups obtained (|**sms**|), the number of calls made to the Gecode solver to obtain sample solutions of either submodel instances or global constraint candidates (**calls**), the number of satisfiability tests performed (**evals**), and some of the global constraints proposed as candidates with a score of 1 (**top candidates**). Note that the output has been manually simplified, with some duplicate or uninteresting constraints removed. This is needed when the system is unable to distinguish between two parameters that appear to be different, but actually refer to the same value. Two data files are used for each model and the default parameters for the number of samples generated (30), and the maximum number of constraints per submodel (2), are used. The set of problems used in the table is as follows:

Cars: a version of the car sequencing problem of 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. Globalizer finds two constraints: `sliding_sum` and `gcc` (global-cardinality).

Jobshop: 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. Globalizer finds a single constraint: the `unary` scheduling constraint.

Party: the running example from Listing 1. Globalizer finds 5 different constraints. The most important ones were discussed earlier: the `bin-packing`, `alldifferent`, and `channel` global constraints. It also finds a `unary` scheduling constraint.

Party-CSP: a slight variant of the Party model, where the constraints are not grouped together under conjunctions (`/\`) under `forall` loops. It also includes the `alldifferent` constraint on the `hostedBy` variables. The results presented in the table show that this has little impact on the amount of time it takes to discover globals. However, we also see that only 3 constraints are found, compared to the 5 of the first model.

Party-LP the integer model presented in Listing 8. Globalizer first detects and introduces integer variables that channel to `visits` named `visits_2of3`. These correspond to the `hostedBy` variables from Listing 1. It then successfully detects the `alldifferent` and `bin_packing_capa` constraints on the `visits_2of3` variables.

Packing: a packing problem that aims to pack `n` squares into a rectangle. The model was taken from the MiniZinc distribution. Globalizer finds `diffn` constraints that express the non-overlapping of the squares.

Schedule: a scheduling example from [16]. The schedule is constrained in such a way that one task needs to start at every even time point, which is equivalent to a `gcc` (global-cardinality) constraint with argument `[1,0,1,0,...]`. Globalizer finds this constraint if all input data files 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 LP: a model of the Sudoku puzzle (where values in each of the nine rows, the nine columns and the nine 3×3 blocks must be distinct) that uses the standard zero-one integer program formulation. Globalizer finds 26 constraints, which include some of the `alldifferent` constraints on the rows and columns. It does not however detect the constraints on the blocks, since the system does not attempt to generate complex array slices. This issue is discussed in more detail in Section 8. In addition to the `alldifferent`

constraints, a set of semantically identical global-cardinality constraints are also detected. These state that the values $1,2,\ldots 9$ can only occur in each row or column a single time.

Sudoku CSP: a model of the Sudoku puzzle represented using a single `forall` loop over every pair of variables in the matrix. A complicated `if` statement then decides whether a not-equal constraint should be posted between each pair or not. Here, loop unrolling is not strong enough to generate candidates that correspond to individual rows, columns, or blocks. As a result, no replacement global constraints is found. This model was designed as a pathological bad case for Globalizer. It motivates somewhat the use of a syntactic matching approach in conjunction with Globalizer, since such an approach might have been able to detect the global constraints in this case.

Warehouses: the warehouse allocation problem of CSPLib 034, which was previously distributed with MiniZinc (version 1.6). Globalizer finds that a loop containing counting constraints can be aggregated into one `gcc` constraint.

The experimental results indicate that the default implementation is quite accurate, as in most cases (e.g., Cars, Party, Packing) Globalizer is able to find the global constraints that an expert modeller would have used. For the rest (e.g., Sudoku CSP), it mainly found constraints with low rank or simply nothing.

The results also show that the approach can be time consuming. The time to analyse a reasonably complex model with 3-4 data sets is in the range of several 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. However, this is not unacceptably slow, as it should not be necessary to use Globalizer often and, if it can find global constraints for the user's model, it can potentially speed up the solving of all resulting instances, which more than makes up for the initial investment. Further, the current implementation is not optimized for performance. There is still great potential for improving performance by, for example, avoiding unnecessary candidate checks and parameter instantiations which, as indicated, make up the bulk of the run time.

The number of generated submodel instance groups ($|\mathbf{sms}|$) is relatively small (usually less than 100), which means 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 humans.

Looking at the number of satisfiability tests, which can reach hundreds of thousands, it is clear that each check needs to be very efficient. This justifies the

problem	time	|cs|	|sms|	calls	evals	top candidates
Cars	31	3	33	1113	73927	`gcc(step_class,cars_in_class)` `sliding_sum(0, option_max_per_block[p], option_block_size[p], step_option_use[*,p])`
Jobshop	111	1	93	4353	410840	`unary(s[*,machine],d[*,machine])`
Packing	96	2	163	4226	329453	`diffn(x,y,pack_s,pack_s)`

Party	438	5	280	7273	211457	`bin_packing_capa(capacity, hostedBy[*,t],crew)`
						`unary(hostedBy[g,*],visits[g,h,*])`
						`alldifferent(hostedBy[g,*])`
						`channel_2of3(hostedBy,visits)`
						`channel(hostedBy[g,t],visits[g,*,t])`

Party CSP	430	3	258	7010	325525	`bin_packing_capa(spareCapacity, h[*,t],crew)`
						`channel_2of3(h,v)`
						`channel(h[i,t],v[i,*,t])`

Party LP	54	5	79	1364	30770	`unary(gamma_1of3[k,*],gamma[i,k,*])`
						`alldifferent(gamma_1of3[k,*])`
						`gamma_1of3 = binaries_represent_int_1of3(gamma)`
						`bin_packing_capa(K,gamma_1of3[*,t],c)`

Schedule	32	1	42	1362	98856	`gcc(x,[1,0,1,0,1,0,1])`
Sudoku CSP	55	0	35	2113	99104	

Sudoku LP	2386	25	556	10826	302905	`alldifferent(x_1of3[j,*])`
						`alldifferent(x_1of3[*,k])`
						`alldifferent(x_2of3[i,*])`
						`alldifferent(x_2of3[*,k])`
						`alldifferent(x_3of3[i,*])`
						`alldifferent(x_3of3[*,j])`
						`x_1of3 = binaries_represent_int_1of3(x)`
						`x_2of3 = binaries_represent_int_2of3(x)`
						`x_3of3 = binaries_represent_int_3of3(x)`
						`diffn(givens[*,j],givens[i,*], x[i,j,*],x[i,j,*])`
						`unary(givens[i,*],x[i,j,*])`
						`unary(givens[i,*],x[i,*,k])`
						`unary(givens[*,j],x[i,j,*])`
						`unary(givens[*,j],x[*,j,k])`
						`unary(x_1of3[*,k],x[i,*,k])`
						`unary(x_2of3[i,*],x[i,j,*])`
						`unary(x_3of3[i,*],x[i,*,k])`

Warehouses	59	2	93	2310	219903	`gcc(supplier,use)`

Table 2: Evaluation of the default configuration of 30 samples and a maximum subproblem size of two constraints on hand-picked models with two data files.

introduction of the dedicated satisfiability evaluator mentioned in Section 6.1.

Analysing Benchmark Models. In addition to these hand picked models, we have also tested Globalizer on a larger suite of models. For these experiments we use

all models from the MiniZinc challenge competition [29] from every year between 2010 and 2018. For each model and year, two data files were selected to be analysed by Globalizer. In total this experiment included 143 separate analyses covering 88 distinct models, as models are often used in multiple challenges with different datasets. It is important to note that since Globalizer was implemented for only a subset of the MiniZinc language, it cannot analyse every model in the benchmark suite. Errors were reported for parts of 40 models. However, the implementation was still able to discover global constraints in some of these models. Table 3 presents some results from this larger experiment, where Globalizer found valid, non-trivial global constraints. Of the models presented here, Globalizer had some trouble with sugiyama, depot-placement, and filters yet it was still able to discover some valid global constraints.

Table 3 shows Globalizer found some interesting constraints (annotated with *). For example, ghoulomb was designed to be difficult for CP solvers. One of the complications in this model is the use of the `cumulative` global constraint to encode an `alldifferent` constraint. Globalizer accurately detects these `alldifferent` constraints. While the replacement does not improve performance it does improve clarity. In prize-collecting a constraint can be replaced with an `alldifferent_except_0` constraint. This replacement had little impact on the solving performance of Gecode. However, the replacement resulted in a significant performance improvement (28% quicker on average) for the learning solver Chuffed [30]. For the sugiyama model, Globalizer suggests an `alldifferent` constraint on an array: `position`. In the model, this array is partitioned into several "Layers", where the set of values available for each layer are disjoint. The proposed `alldifferent` constraint replaces a set of `alldifferent` constraints constraining the values within a layer to be distinct. This replacement is equivalent, but it does not provide any extra propagation. Additionally, the meaning of the replacement is less obvious than the explicit set of `alldifferent` constraints, reducing the readibility of the model.

Along with these interesting cases, Globalizer also suggested many incorrect or not particularly useful constraints (annotated with *dagger*), which highlight some of the limitations of our current implementation. For example, for cyclic-rcpsp and stochastic-fjsp

problem	time	|cs|	|sms|	calls	evals	top candidates
amaze	456	1	365	7645	51112	`element(end[i],next,end[i])`
carpet-cutting	2142	1	1327	10915	36662	`channel(0,rm_x)`
cyclic-rcpsp	833	2	166	2583	16251	†`lex_lesseq(k,s)` †`lex_less(k,s)`

depot-placement	1018	9	2442	27614	2543763	`channel(0,ALegDist)`
						`channel(0,BLegDist)`
						`channel(ALegDist[ASize],ALegDist)`
						`element`

filters	105	10	116	2468	17699	`count(r,del_add,n)`
						`count(r,number_add,n)`
						`count(r,number_mul,n)`
						`element`
						†`maximum(1,r)`
						†`maximum(del_add,r)`
						†`maximum(number_add,r)`
						†`maximum(number_mul,r)`

ghoulomb	237	6	208	3333	13853	*`alldifferent(mark1)` *`alldifferent(mark2)` †`member(mark1,0)` †`member(mark2,0)` †`minimum(0,mark1)` †`minimum(0,mark2)`

league	617	5	503	18060	172215	`count(assign_to,i,n_persons[i])`
						†`increasing(max_rank)`
						`increasing(min_rank)`
						`sort(max_rank,max_rank)`

mario	86	1	34	752	4766	`element`
maximum-dag	21	1	16	20	89	`element`
open-stacks	352	3	47	1189	5842	`gcc(s,[1,1,...,1,1])` †`member(o[i,..],0)` †`minimum(0,o[i,..])`
prize-collecting	200	2	150	3031	18868	*`alldifferent_except_0(next)` *`alldifferent_except_0(next)`
roster	316	6	195	6060	44434	`count(roster[..,day],shift,reqt[shift,day])` `gcc(roster[..,day],reqt[..,day])`

stochastic-fjsp	201	9	147	5993	36250	`element`
						†`lex_lesseq(machine,start_time[sc,..])`
						†`lex_less(machine,start_time[sc,..])`
						†`member(duration,machine[t])`
						`sliding_sum(ta,additional_tasks_end[sc,ta], additional_tasks_machine[sc,ta], machine)`

sugiyama	592	2	417	7463	18479	*`alldifferent(positions)`
tpp	73	3	104	1858	13408	`element` †`member(succ,succ[numcities])`
triangular	2850	2	22	1053	2621	`element`

vrp	3016	6	135	3556	21239	`binaries_represent_int(x[i,..])`
						`binaries_represent_int(x[..,j])`
						`count(x[i,..],0,N)`
						`count(x[..,j],0,N)`

Table 3: Evaluation of the default configuration of 30 samples and a maximum subproblem size of two constraints on a wider set of benchmark models with two data files.

Globalizer suggested some lex_less constraints. While these are valid constraints, they are not very useful as the two arrays of variables involved represent very different types of objects (this is most obvious looking at the constraints suggested for stochastic-fjsp where a lexicographic ordering is being enforced between machines and task start times). Since Globalizer was first implemented, the MiniZinc language has added support for enumerated types. Using enumerated types, Globalizer could avoid making such mistakes, as it would recognise that the variables are different types, rather than just seeing them as plain integer variables.

In some cases, Globalizer also produces spurious suggestions (the minimum, maximum, and member constraints). These are valid only in the context of a small part of the model, and appear because candidates are not tested in the context of the full constraint model. This limitation comes from the assumption that it may be too expensive to test against the full model too frequently.

7.2. Evaluating Alternative Configurations

Let us now evaluate the effect of varying the number of data files per problem used to instantiate submodels (2 in the default configuration), the number of sample solutions used to build the templates and filter the global constraint candidates (30 in the default configuration), and the maximum number of constraints in the submodels (2 in the default configuration), on the accuracy and practicality of the method. Table 4 shows the results of this evaluation, which needs to be considered together with those presented in Table 2. The columns **time**, |**cs**|, |**sms**|, **calls**, and **evals** represent the same information as before. Figure 3 shows plots demonstrating in graphical form several interesting results from these experiments. The plotted values are ratios, with the first configuration being the baseline.

The results shown in the first row of Table 4 (together with those in Table 2) indicate that increasing the number of data files can reduce the number of global constraint candidates generated. For example, analysing the Sudoku LP problem with a single data file results in 29 candidate global constraints with a score of 1. The number of candidates generated is reduced to 25 with 2 data files (shown in Table 2), and 24 with 3. Interestingly, for many problems a single data file was enough to generate a number of candidates that was not reduced by adding more data files. Figure 3a (which includes the results from Table 2) demonstrates this point graphically. In fact, the addition of a third data file results in a further reduction only in the case of the Sudoku LP model. The overhead required for this reduction can be seen in Table 4 where we see that increasing the number of data files always increases the amount of time required. Two data files were selected for our default, as a middle-ground.

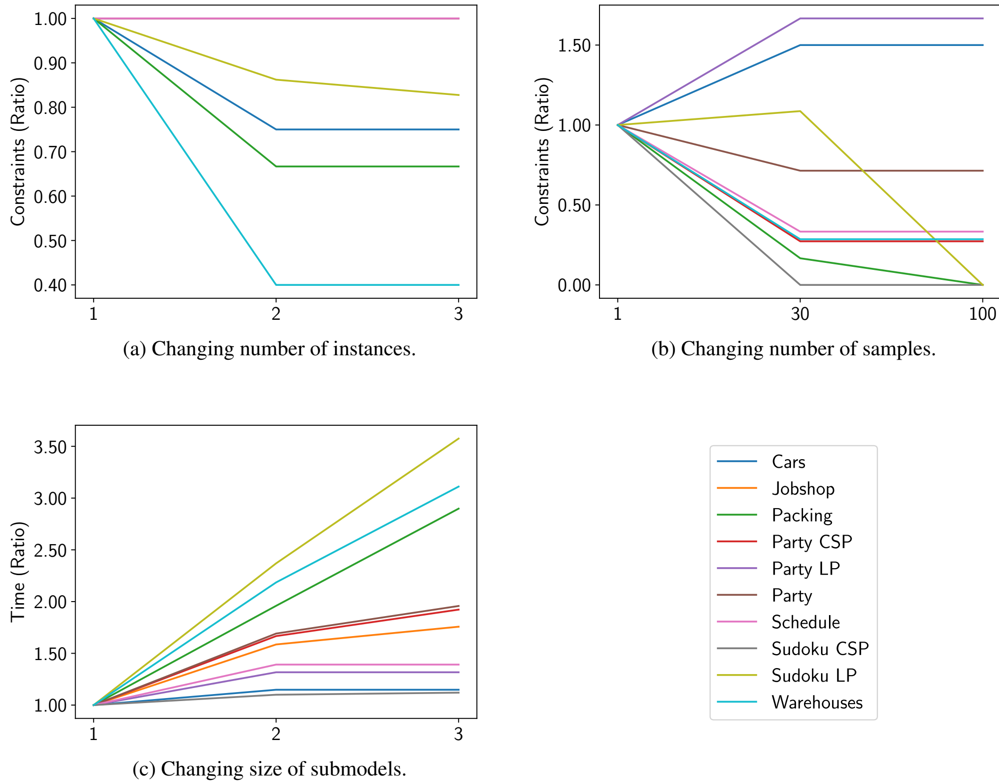
	time	|cs|	|sms|	calls	evals	time	|cs|	|sms|	calls	evals
problem	1 Datafile					3 Datafiles				
Cars	24	4	33	810	72,561	39	3	33	1,428	73,535
Jobshop	95	1	93	3,783	400,466	127	1	93	4,916	409,086
Packing	57	3	89	2,431	223,813	111	2	163	5,090	324,279
Party	328	5	280	4,475	213,990	524	5	280	10,076	210,556
Party CSP	343	3	258	4,568	332,407	499	3	258	9,479	319,538
Party LP	36	5	79	789	27,293	69	5	79	1,892	32,965
Schedule	26	1	42	1,056	98,933	38	1	42	1,657	99,847
Sudoku CSP	39	0	35	1,411	97,855	70	0	35	2,803	100,704
Sudoku LP	1,673	29	556	6,518	268,389	3,092	24	556	15,126	336,689
Warehouses	45	5	93	1,761	219,828	75	2	93	2,920	223,148
	1 Sample					100 Samples				
Cars	26	2	20	1,028	15,938	39	3	33	1,101	174,168
Jobshop	67	0	19	2,087	21,714	157	1	93	5,359	1,405,861
Packing	77	12	89	3,266	40,310	118	0	163	4,359	835,164
Party	156	7	120	5,594	42,482	1,012	5	280	7,216	502,975
Party CSP	168	11	167	6,070	76,704	991	3	258	6,963	700,184
Party LP	32	3	79	1,148	4,013	98	5	79	1,367	87,673
Schedule	49	3	42	2,105	31,526	42	1	42	1,453	254,423
Sudoku CSP	68	2	35	2,969	36,096	81	0	35	2,116	250,341
Sudoku LP	419	23	556	10,711	47,460	3,601	0	556	0	0
Warehouses	99	7	93	4,167	101,783	67	2	93	2,231	444,969
	1 Constraint					3 Constraints				
Cars	27	3	27	964	61,024	31	3	34	1,115	73,470
Jobshop	70	1	60	2,404	212,129	123	1	105	4,847	463,539
Packing	49	2	78	2,237	146,203	142	2	243	6,200	514,126
Party	259	5	140	4,078	112,454	507	5	352	8,429	240,307
Party CSP	258	3	130	4,000	167,774	496	3	327	8,119	374,907
Party LP	41	5	52	1,072	24,478	54	5	87	1,374	31,464
Schedule	23	1	30	958	66,213	32	1	43	1,384	100,715
Sudoku CSP	50	0	34	1,932	96,259	56	0	35	2,101	98,771
Sudoku LP	1,007	24	192	4,897	103,959	3,600	0	1,036	0	0
Warehouses	27	2	45	1,074	73,241	84	2	126	3,284	346,203

Table 4: Comparison of different configurations.

As shown in the second row of Table 4 (plus Table 2), and graphically in Figure 3b, increasing the number of samples tends to reduce the number of spurious global constraint candidates found, by eliminating those that meet the conditions simply by chance. Since the number of constraints found using 30 samples and 100 samples are quite similar, we chose 30 for our default implementation. Note, however, that increasing the number of samples results in a roughly linear increase in running time, which seems a reasonable price to pay for the increased accuracy.

An interesting, non-intuitive result can be found in the progressive party model. Here we find a `cumulative` constraint with the arguments: `hostedBy[_, t]`, `visits[_, h, t]`, `crew`, and `capacity[h]`, which encodes:

```
sum (g in GuestCrews) (crew[g]*visits[g,h,t]) <= capacity[h]
```



(a) Changing number of instances.

(b) Changing number of samples.

(c) Changing size of submodels.

Figure 3: Experiment looking at different configurations.

This plays a similar role to the bin-packing constraint, but the use of `cumulative` requires a different perspective: instead of packing objects (guest crews) into capacitated bins (host boats), we are scheduling tasks (guest crews) such that the resource (host boat capacity) is not exhausted, and the duration of each "task" is 1 if the crew visits the host in that time slot and zero otherwise. The `cumulative` constraint is eliminated in the 100 sample run because an edge case (all tasks with zero duration) is not permitted by the definition of `cumulative`, and the edge case is only struck upon with the larger sample size. This demonstrates Globalizer's ability to present alternative viewpoints to the modeller, in this case presenting this subproblem as a scheduling task rather than as a bin-packing task.

Figure 3c shows graphically, that increasing the number of constraints in a submodel (increasing the size) has the expected effect of increasing the runtime. While for many models the increase is sub-linear, in some cases it can really blow up, with the Sudoku LP model (the top line in this plot) taking more than three

times longer when submodels can contain up to 3 constraints. This is why the default configuration uses 2 constraints.

8. Limitations and Future Work

This section explores some of the main limitations of the current Globalizer approach and possible directions for future work.

Efficiency. Currently, globalizing a model using Globalizer can be a slow process. While we believe the cost is still reasonable given the potential speed-ups when solving its instances, the globalizing time can be reduced. For example, the implementation of heuristics that would prioritise the order in which submodels are explored, could allow global constraints to be discovered earlier. Similarly, the system could test whether a constraint occurs in a subset of a set of constraints that have already been explored. However, it is unclear how much of a performance boost the pruning of these redundant submodels would provide.

Stronger ranking and filtering. This could be achieved by incorporating the ranking techniques from Constraint Seeker, such as using known implications between constraints to eliminate more imperfect candidates. Model Seeker could also be used to generate more complex candidates for each submodel. The confidence in the suggested constraints may also be improved by using theorem proving or other techniques to prove equivalence in cases where it is possible.

Array slicing. A natural application of the tool is in teaching combinatorial problem modelling and solving. In a recent assignment given as part of such a course at Monash University, students were required to write a MiniZinc model to solve a vehicle routing problem. The problem involved determining a tour to deliver goods to customers over several days using several trucks, where each customer has some fixed demand and each truck some fixed capacity. A constraint of the problem is that the trucks cannot be overloaded. Some students modelled this as shown in Listing 9. Those constraints can be replaced by a `bin-packing` constraint, where the trucks are the bins, and the customers are the objects to be placed in the bins. Note that the tour length has more entries than the number of customers, because it also includes the start and end visits to the depot. The replacement constraint could be written as:

```
bin_packing_capa(Capacity,
                 [Assignments[i] | i in 1..NumCustomers],
                 Demand);
```

```
% Parameters
array[1..NumCustomers] of int: Demand;
array[1..NumVehicles] of int: Capacity;
int : TourLength = NumCustomers + 2*NumVehicles
% Assignment of customers / depots to vehicles
array[1..TourLength] of var 1..NumVehicles: Assignments;
...
% Demand/capacity constraint
constraint forall(v in Vehicles) (
    sum(i in 1..NumCusts)
       (Demand[i]*(Assignments[i]==v)) <= Capacity[v]);
```

Listing 9: One way to model that tracks should not be overloaded

However, this constraint cannot be found by Globalizer, as it does not construct "array slices" (as done by the second argument of `bin_packing_capa`) as potential arguments. A slice of the array is necessary because the `Demand` array is smaller than the `Assignments` array, as the `Assignments` array also contains the visits to the depots, and for the `bin-packing` we want to consider only the visits to the customers. We also saw this earlier in Table 2 where for the Sudoku LP model the `alldifferent` constraints on sub-squares of the matrix could not be found.

An approach to detect this would be to modify Globalizer to generate every possible array slice as a possible argument. However, this can be too expensive. Instead, we can modify Globalizer to analyse syntactically the array access expressions in the constraint to determine which elements of the array are used, and use this subset as a potential argument. We expect this to work in most cases. In the example above, we can see that the `Assignments` array is only used with `i` as an argument, and `i` is declared to be drawn from the set `1..NumCusts`. Via this simple syntactic analysis, we can see that the constraint acts only on the subset of the array given by `[Assignments[i] | i in 1..NumCustomers]`. Therefore, we can consider this slice of the `Assignments` array as a potential argument.

Higher-order types. While the method presented in this paper could be adapted for the modelling language ESSENCE [1], further extending the approach to support its higher-order types would likely require more work. It is not clear to us whether a method similar to Globalizer could be used to suggest, for example, how to re-model a problem using sets of multisets rather than a low-level decomposition into individual Boolean variables. Note that this kind of analysis is similar to the detection of channeling constraints as discussed in Sect. 2.2, but it may not be

trivial to extend it to more complex types.

MiniZinc language compatibility. As mentioned before, Globalizer currently supports only a subset of the full MiniZinc language. We are planning to add the main missing features, such as support for set and optional variables as well as enumerated types, in a future release.

9. Conclusions

This paper proposes a method for *globalizing* combinatorial problem models, that is, for inferring candidate global constraints that might be able to replace parts of a given model of a combinatorial problem. 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 combinatorial solving technology and faster solving using specialised algorithms. This in turn can yield higher quality solutions to the combinatorial decision problems that occur in many areas of our society, from health and transport, to energy and human resources.

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 then ranked and filtered based on how well their search spaces match.

Importantly, the same approach can be applied as an initial pass that focusses on finding common alternative viewpoints (often referred to as duals) in the form of binary variables that may represent integer variables. These viewpoints can then be added to the model so that, in the second pass and with the full set of constraints, Globalizer may be able to find candidate constraints on the variables of these alternative viewpoints.

Experimental evidence was presented showing that the method can be both practical and accurate. Experiments were also presented that explore different implementation configurations. The implementation of Globalizer for MiniZinc models has been released as part of the MiniZinc distribution available from `https://www.minizinc.org/software.html`.

References

[1] A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel, Essence: A constraint language for specifying combinatorial problems, Constraints 13 (3) (2008) 268–306.

[2] K. Marriott, N. Nethercote, R. Rafeh, P. Stuckey, M. Garcia de la Banda, M. Wallace, The design of the Zinc modelling language, Constraints 13 (3) (2008) 229–267.

[3] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: C. Bessiere (Ed.), Principles and Practice of Constraint Programming-CP, Vol. 4741 of LNCS, Springer, 2007, pp. 529–543.

[4] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A fast dynamic language for technical computing, arXiv preprint arXiv:1209.5145 (2012).

[5] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006.

[6] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: CP, Vol. 1520 of LCNS, Springer, 1998, pp. 417–431.

[7] W. Winston, M. Venkataramanan, J. Goldberg, Introduction to mathematical programming, Vol. 1, Thomson/Brooks/Cole, 2003.

[8] G. Chu, P. J. Stuckey, Minimizing the maximum number of open stacks by customer search, in: Principles and Practice of Constraint Programming - CP 2009, Springer, 2009, pp. 242–257.

[9] K. Leo, G. Tack, Multi-Pass High-Level Presolving, in: Q. Yang, M. Wooldridge (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, AAAI Press, 2015, pp. 346–352.

[10] P. Van Hentenryck, P. Flener, J. Pearson, M. Âgren, Compositional derivation of symmetries for constraint satisfaction, in: Abstraction, Reformulation and Approximation, Vol. 3607 of LNCS, Springer, 2005, pp. 234–247.

[11] B. M. W. Cheng, J. H. M. Lee, J. C. K. Wu, Speeding up constraint propagation by redundant modeling, in: Principles and Practice of Constraint Programming, Second International Conference, CP 1996, Proceedings, 1996, pp. 91–103.

[12] A. Frisch, I. Miguel, T. Walsh, Extensions to proof planning for generating implied constraints, in: Calculemus-01, 2001, pp. 130–141.

[13] A. M. Frisch, I. Miguel, T. Walsh, CGRASS: A system for transforming constraint satisfaction problems, in: Recent Advances in Constraints, Vol. 2627 of LNCS, Springer, 2003, pp. 15–30.

[14] I. P. Gent, I. Miguel, A. Rendl, Tailoring solver-independent constraint models: A case study with Essence' and Minion, in: I. Miguel, W. Ruml (Eds.), Abstraction, Reformulation, and Approximation, Vol. 4612 of LNCS, Springer, 2007, pp. 184–199.

[15] A. M. Frisch, C. Jefferson, B. Martínez-Hernández, I. Miguel, The rules of constraint modelling, in: International Joint Conference on Artificial Intelligence, Vol. 19, Lawrence Erlbaum Associates LTD, 2005, pp. 109–116.

[16] C. Bessiere, R. Coletta, T. Petit, Learning implied global constraints, in: M. M. Veloso (Ed.), IJCAI, 2007, pp. 44–49.

[17] J. Charnley, S. Colton, I. Miguel, Automatic generation of implied constraints, in: European Conference on Artificial Intelligence-ECAI, Vol. 141, IOS Press, 2006, pp. 73–77.

[18] N. Beldiceanu, H. Simonis, A model seeker: Extracting global constraint models from positive examples, in: M. Milano (Ed.), Principles and Practice of Constraint Programming-CP, Vol. 7514 of LNCS, Springer, 2012, pp. 141–157.

[19] N. Beldiceanu, H. Simonis, A constraint seeker: Finding and ranking global constraints from examples, in: J. H.-M. Lee (Ed.), Principles and Practice of Constraint Programming-CP, Vol. 6876 of LNCS, Springer, 2011, pp. 12–26.

[20] K. Leo, C. Mears, G. Tack, M. G. de la Banda, Globalizing constraint models, in: C. Schulte (Ed.), Principles and Practice of Constraint Programming

- 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings, Vol. 8124 of Lecture Notes in Computer Science, Springer, 2013, pp. 432–447.

[21] R. Fourer, D. M. Gay, B. W. Kernighan, Ampl: A mathematical programming language, in: S. W. Wallace (Ed.), Algorithms and Model Formulations in Mathematical Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 150–151.

[22] B. M. Smith, S. C. Brailsford, P. M. Hubbard, H. P. Williams, The progressive party problem: Integer linear programming and constraint programming compared, Constraints 1 (1) (1996) 119–138.

[23] N. Beldiceanu, M. Carlsson, S. Demassey, T. Petit, Global constraint catalogue: Past, present and future, Constraints 12 (1) (2007).

[24] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan, A SAT-based version space algorithm for acquiring constraint satisfaction problems, in: Machine Learning: ECML 2005, Springer, 2005, pp. 23–34.

[25] C. Bessiere, F. Koriche, N. Lazaar, B. O'Sullivan, Constraint acquisition, Artif. Intell. 244 (2017) 315–342.

[26] Ö. Akgün, Extensible automated constraint modelling via refinement of abstract problem specifications, Constraints 22 (1) (2017) 91–92. `doi:10.1007/s10601-016-9258-6`.
URL `https://doi.org/10.1007/s10601-016-9258-6`

[27] P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, P. Spracklen, Automatically improving constraint models in savile row, Artif. Intell. 251 (2017) 35–61. `doi:10.1016/j.artint.2017.07.001`.
URL `https://doi.org/10.1016/j.artint.2017.07.001`

[28] Gecode Team, Gecode: Generic constraint development environment (2006).
URL `http://www.gecode.org`

[29] P. Stuckey, R. Becket, J. Fischer, Philosophy of the MiniZinc challenge, Constraints 15 (3) (2010) 307–316. `doi:{10.1007/s10601-010-9093-0}`.

[30] G. Chu, Improving combinatorial optimization, Phd thesis, Department of Computing and Information Systems, University of Melbourne (2011).