Improved Linearization of Constraint Programming Models

Gleb Belov[1], Peter J. Stuckey[2], Guido Tack[1], and Mark Wallace[1]

[1] Monash University, Caulfield Campus, Australia
{name.surname}@monash.edu
[2] Data61, CSIRO and University of Melbourne, Parkville, Australia
pstuckey@unimelb.edu.au

Abstract. Constraint Programming (CP) standardizes many specialized "global constraints" allowing high-level modelling of combinatorial optimization and feasibility problems. Current Mixed-Integer Linear Programming (MIP) technology lacks both a modelling language and a solving mechanism based on high-level constraints.
MiniZinc is a solver-independent CP modelling language. The solver interface works by translating a MiniZinc model into the simpler language FlatZinc. A specific solver can provide its own redefinition library of MiniZinc constraints.
This paper describes improvements to the redefinitions for MIP solvers and to the compiler front-end. We discuss known and new translation methods, in particular we introduce a coordinated decomposition for domain constraints. The redefinition library is tested on the benchmarks of the MiniZinc Challenges 2012–2015. Experiments show that the two solving paradigms have rather diverse sets of strengths and weaknesses. We believe this is an important step for modelling languages. It illustrates that the high-level approach of recognizing and naming combinatorial substructure and using this to define a model, common to CP modellers, is equally applicable to those wishing to use MIP solving technology. It also makes the goal of solver-independent modelling one step closer. At least for prototyping, the new front-end frees the modeller from considering the solving technology, extracting very good performance from MIP solvers for high-level CP-style MiniZinc models.

Keywords: combinatorial optimization, high-level modelling, automatic reformulation, linear decomposition, context-aware reformulation

1 Introduction

Constraint Programming (CP) operates in terms of specialized constraints, from basic ones such as arithmetic, to high-level "global constraints" [3], and their filtering/explanation algorithms. A solver which handles a model's high-level structure in terms of global constraints, can take advantage of this knowledge in different ways. When the solver does not provide a handler for a certain constraint, the latter can be expressed by more basic entities.

Current Mixed-Integer Programming (MIP) technology lacks a modelling language based on global constraints, so that a dedicated MIP modeller has to hard-code a chosen MIP decomposition of his real problem. Moreover, a MIP solver might need to reverse-engineer the high-level model information for efficiency [24].

We consider automatic linearization of CP models, producing good quality MIPs. MiniZinc [21] is a solver-independent Constraint Programming modelling language. A MiniZinc model is translated into FlatZinc, a low-level language, and the translation is controlled by a redefinition library. The MiniZinc front-end is now supported by some 20 solvers, including finite domain solvers, SAT solvers, Lazy Clause Generation solvers, and even local search solvers [4]. Annual MiniZinc competitions [27] provide a basis for comparing solvers and exploring their strengths and weaknesses.

A number of modelling front-ends are available for MIP solvers, including GAMS [2] and AMPL [10] which focus on general Mathematical Programming. For combinatorial problems, AMPL supports logical constraints and counters. A similar functionality is offered by ZIMPL [15]. There are a number of CP languages offering automatic translation to MIP [1].

Our vision is that MiniZinc becomes an accepted and even widely-used modelling language within the OR community, thus helping to narrow the divide between OR, CP and SAT researchers, and to simplify prototyping. To this purpose we seek to ensure that pure MIP models, when formulated in MiniZinc, have similar performance to AMPL and GAMS. This requires no reformulation, but care needs to be taken in user and solver interfaces. Beyond that, we try to optimize the MIP-compatible reformulation of CP models to make it flexible and extensible. This enables the modeller to use the CP style modelling where combinatorial substructure is captured using global constraints, and obtain good performance for their problem, using state-of-the-art MIP, CP and SAT solving technologies on the same models.

In the MiniZinc Challenge [27] MIP solvers have had some success, but MIP did not appear competitive on most of the Challenge benchmarks. Some models are inherently more efficient for MIP solving, e.g., assignment problems (see the example in Section 2.2), and problems involving network flow. But we were suspicious that the relatively poor performance of MIP solvers was an artifact of a naive transformation of CP models to MIP. By improving linearization we can see the true potential of MIP solvers on the Challenge benchmarks.

In automatic reformulation, it is up to the modelling system to provide efficient translation for the target solver. To yield efficient transformed models it is important to ensure that auxiliary variables generated during reformulation are not unnecessarily duplicated [23]. Cire et al [7] define an interactive system that aids automatic detection of equivalent auxiliary variables produced in reformulations of various parts of a model. MiniZinc 2.0 takes a different approach through user-defined functions [28] which are used to avoid duplication in the first place.

Refalo [23] presents a system for automatic reformulation of global constraints into MIPs. He observes that the reformulations are usually standard and tight. The system supports dynamic reformulation: as more information about the model becomes available during solving, the reformulation is updated. However, the implementation is bound to a hybrid of specific CP and MIP solvers.

Among the "basic" non-linear constraints we consider domain constraints, restricting an integer variable to take values in a specified set, both in static as well as reified version (i.e., depending on another condition). Such constraints can appear on their own in a model, or be produced by the decomposition of other non-linear constraints, such as disjunction, array element access, and many others. We propose a coordinated decomposition of domain constraints which takes into account all those of a group of dependent variables.

The next section gives introductory examples. Section 3 discusses general linearization methods, in particular introducing the new domain constraint decomposition. Experimental results follow.

2 Basics and Redefinition Examples

This section provides an overview of MiniZinc's redefinition mechanism and some motivating examples.

2.1 Basics on MiniZinc and Solver-Specific Redefinitions

MiniZinc [28] is a declarative modelling language. It builds constraint structures using *predicates*, here is a toy example:

```
predicate small(int: m, var int:y) =  -m <= y /\ y <= m;
predicate p(int: u, var bool: b, var int: x) =
          (b <-> small(u,x));
constraint p(4,false,v);
```

Global constraints [3], such as the well known `alldifferent`, are also specified as predicates.

When the model is compiled for a specific solver, the front-end looks for a solver-specific redefinition of the global constraints used. If none is provided, MiniZinc has a default decomposition, for example the standard library definition for `alldifferent` is:

```
predicate alldifferent(array[int] of var int: x) =
       forall(i,j in index_set(x) where i < j)
         ( x[i] != x[j] );
```

However the solver can provide its own redefinition. In particular, it can forward the predicate call unchanged and use specialized algorithms. The translated model is converted to the low-level FlatZinc format and can be passed to the solver, or used directly if the solver is linked in the same executable.

For example, the `alldifferent` constraint can be redefined in a different way than given above for a linear solver:

```
predicate alldifferent(array [Set1] of var Set2: x)  =
    forall (j in Set2)( sum(i in Set1)(x[i]==j) <= 1 );
```

This redefinition automatically introduces an auxiliary zero-one variable to encode the assignment of a variable to a value, `x[i]==j`, see Section 3. Note that this auxiliary variable is *re-used* whenever this equality is encoded again, see Section 3.1.

The redefinition library for MIP is located in folder `share/minizinc/linear` of the MiniZinc distribution. To use this library, the `mzn2fzn` compiler is called with options `-G linear`. In particular, files `redefinitions*.mzn` re-define the basic constraints, such as logical connectives and min/max. Most global constraints are specified in dedicated files, for example `lex_less.mzn`. If a library does not provide a header for some global, its default decomposition is taken from the standard library `share/minizinc/std`.

2.2 Linearization Example: Assignment Problem

Consider an assignment problem. Its natural CP model is:

```
set of int: WORKER ;  % workers
set of int: TASK ;    % tasks to be assigned to workers
array[WORKER,TASK] of int: value;
array[WORKER] of var TASK: task; % which task worked on by each worker
include "alldifferent.mzn";
constraint alldifferent(task); % each worker works on a different task
solve maximize sum(w in WORKER)(value[w,task[w]]);
```

The natural MIP formulation of the model is the following one:

```
set of int: WORKER ;  % workers
set of int: TASK ;    % tasks to be assigned to workers
array[WORKER,TASK] of int: value;
array[WORKER,TASK] of var 0..1: worker_task;
constraint forall(w in WORKER)         % one task per worker
                 (sum(t in TASK)(worker_task[w,t]) = 1);
constraint forall(t in TASK)
                 (sum(w in WORKER)(worker_task[w,t]) <= 1); % alldifferent
solve maximize sum(w in WORKER, t in TASK)(value[w,t] * worker_task[w,t]);
```

Unsurprisingly the MIP solver is effectively ”infinitely” faster than a CP solver on this problem since the MIP solver will effectively implement a polynomial-time maximal matching algorithm using the linear integer constraints that arise in its formulation. The challenge for the automatic linearization is to ensure that the ”natural” CP model above results in the MIP formulation being sent to the MIP solver, so that we can make use of the insights of combinatorial substructure without being penalized. This is particularly important when we want to solve assignment problems with other side constraints.

As explained in Section 2.1, our automatic linearization of `alldifferent` produces the exact equivalent of its “natural” MIP decomposition. For the objective function, which accesses element `task[w]` in each row `w` of matrix `value`, the compiler transforms nested matrix access `value[w,task[w]]` into a standard array access represented by the global constraint `element` [3]. The latter is linearized as follows [13]:

```
sum (t in TASK) ( value[w,t] * (task[w]==t) )
```

Note that the auxiliary binary variable representing the equality `task[i]==j` is re-used, which altogether gives the natural MIP formulation. There will be some overhead for the (now useless) original `task` variables. However we have an instance of a 3D orthogonal packing model where such variables improve search behavior of IBM ILOG CPLEX 12.6.3 [14].

2.3 Linearization Example: Tour Guide Allocation Problem

An application brought to us by a local company is the tour guide allocation problem. For a set of planned tours with fixed locations and times, the requirement is to minimize the total number of guides needed as well as the travel costs of the guides between their tours.

Let matrix `travel_cost` contain the travel costs between tours and a 4-column matrix `tour` contain start day, duration, start and end location of a tour in each row. Variable array `succ` describes the successors of each tour in its guide's sequence of tours, as follows:

```
int: tour_ct;       % The total number of planned tours
set of int: C = 1..4;  % Columns of tour data structure
int: SDay = 1; int: Dur = 2; int: SLoc = 3; int: ELoc = 4; % Column names
array [1..tour_ct,C] of int: tour;
int: loc_ct;        % Number of locations
array [1..loc_ct, 1..loc_ct] of int: travel_cost;
array [1..tour_ct-1] of var 1..tour_ct: succ;
```

The last tour with index `tour_ct` is the END tour with a zero distance to all other tours' locations. This ensures that in an optimum no two tours have the same successor (different from END).

The total travel cost is the sum of the cost of traveling from the end of each tour to the start of its successor (as recorded by `succ`):

```
constraint   total_travel_cost = sum (t in 1..tour_ct-1)
                (travel_cost[tour[t,ELoc],tour[succ[t],SLoc]]);
```

As in Section 2.2, the nested matrix element accesses are simplified by the compiler, resulting in a linear constraint.

Another array of decision variables is `first_tour`. It tells us how many tour guides are to be used. It does this by selecting certain tours to be the first tour on (some) tour guide's sequence of duties. Then, every tour must have a tour guide (either it must be a first tour or it must be the successor of another tour):

```
array [1..tour_ct-1] of var bool: first_tour;
constraint forall(t in 1..tour_ct-1)
                 ( first_tour[t] \/
                   ( exists (t2 in 1..tour_ct-1) (t = succ[t2]) )
                 ) ;
```

Again, decisions `t==succ[t2]` are converted into auxiliary binary variables, using either unary decomposition or domain refinement (Section 3). These auxiliary variables are automatically the same as in the linearization of the travel cost.

Finally, the successor of tour `t` must have a start date greater than or equal to the start date of `t` plus the duration of `t`:

```
constraint forall(t in 1..tour_ct-1)
                (tour[t,SDay]+tour[t,Dur] <= tour[succ[t],SDay]);
```

The “every tour must have exactly one tour guide” constraint can be made explicit by a direct MIP-tailored network flow-type formulation as follows:

```
constraint forall(t in 1..tour_ct-1)
                (first_tour[t] + sum(t2 in 1..tour_ct-1)(t = succ[t2]) = 1);
```

On an example with 25 locations and 41 tours, CP finds only a suboptimal solution in observable time. Previously it was necessary to write a different MiniZinc model to elicit the efficient performance of a network-flow model with a MIP solver. Now, with automatic linearization, with or without the ”every tour has exactly one tour guide” constraint, IBM ILOG CPLEX 12.6.3 [14] proves an optimum without branching.

3 Linearization

This section discusses some basic linearization principles, introduces domain refinement, and discusses decomposition of the most commonly used global constraints.

3.1 Linearization Principles

Linearization by “Big-M”s. The basic linearization method for complex constraints is the so-called big-M transformation (see e.g. [19, 13]). Given a linear constraint $e \leq 0$ in disjunction with a Boolean b, that is $e \leq 0 \vee b$ or equivalently $\neg b \rightarrow e \leq 0$, then if M is the largest possible value linear expression e can take, this can be expressed using the linear constraint $e \leq Mb$.

For example, $x \neq y$ is equivalent to a disjunction between two inequalities:

$$x \geq y + 1 \;\vee\; y \geq x + 1 \tag{1}$$

which can in turn be transformed by introducing a binary variable b into the conjunction of two implications: $b \rightarrow x \geq y + 1$ and $\neg b \rightarrow y \geq x + 1$, which can then be transformed to linear constraints. Assume x and y range over $[0, 10]$ we can encode the first constraint using the linear constraint $y + 1 - x \leq 11(1 - b)$ and the second by $x + 1 - y \leq 11b$.

Linearization of complex constraints consists of breaking them down into reified linear constraints, and then replacing these with linear constraints using the big-M method illustrated above, or other methods described in this section. For space reasons we don’t describe the MIP decompositions of other basic constraints, such as logical ones, referring the reader, e.g., to [13, 23].

Example 1. Consider the model on the left in Fig. 1. Using “big-M”s, we can linearize the two constraints as shown on the right in the same figure. Its continuous relaxation allows the solution `x==5.5; beta1==beta2==0.75`. □

```
var 0..10: x;
var bool:  beta1;
var bool:  beta2;
constraint beta1 <-> x<=4;
constraint beta2 <-> x>=7;
```

```
x-4 <= 6*(1-beta1)
5-x <= 5*beta1
7-x <= 7*(1-beta2)
x-6 <= 4*beta2
```

Fig. 1. Example model and its “big-M” linearization

Linearization with Unary Encoding of the Domain. An alternative approach to linearization of complex constaints is to introduce a binary variable b_k^x for each value k in the domain $D(x)$ of x [23]. The correspondence between the binary variables and the original integer variable can be enforced by the linear constraints

$$\textstyle\sum_{k\in D(x)} b_k^x = 1, \tag{2a}$$

$$\textstyle\sum_{k\in D(x)} k b_k^x = x. \tag{2b}$$

Unary encoding introduces a lot of auxilliary variables, however it is usually preferred due to its tighter continuous relaxation. There are many constraints which are best transformed using these binary variables, including `alldifferent`, `element` (see [23] and Section 2.2), `inverse`, multiplication of variables, and some others.

Tight Reformulation Using Common Subexpression Elimination. To achieve a tight MIP model without duplicate variables and constraints, it is essential that when a constraint on the same variable is transformed using its unary encoding, the same binaries are used. When the translation is controlled by a library, this can be achieved automatically through MiniZinc’s mechanism of user-definable functions [28].

To introduce these binaries, we use the function `eq_encode(var int: x)` (which was named `int2array` in [28]), returning an array of 0–1 variables and imposing linear constraints (2). Now every time this function is invoked on a variable x, MiniZinc’s common subexpression elimination ensures that the same binaries are reused, even if the function is embedded in a predicate or another function.

However there still can be information loss. For example, $x \neq y - 5$ or $y \neq x$ would be linearized using unary encodings of variables $z' = x - y + 5$ and $z'' = y - x$, respectively. The current capabilities of the MiniZinc language do not allow it to recognize that we could make use of the same unary encoding for these cases and we tackle this issue together with unified domain refinement in Section 3.2.

Multiplication. In MiniZinc 1.6, the decomposition for FlatZinc predicate `int_times` constraining $z = xy$ was $z = (xy_{\min}, \ldots, xy_{\max})_{y-y_{\min}+1}$, or equivalently, using explicit calls to the global constraint `element` [3], `element`$(y -$

$y_{\min}+1, [xy_{\min}, \ldots, xy_{\max}], z)$, where $y_{\min}, y_{\max}$ are the finite bounds of y. Note this method will also work when x is real-valued.

In the cases of a small (chosen as 4..20) product domain size $|D(x)| \times |D(y)|$ and no variable domain having the form $\{0, k\}$, $k \in \mathbb{Z} \setminus \{0\}$, experiments proved that it is advantageous to use the following alternative encoding:

$$z = \textstyle\sum_{i,j} i \times j \times b^{xy}_{ij}, \qquad \text{where} \quad b^{xy}_{ij} = 1 \leftrightarrow (b^x_i = 1 \wedge b^y_j = 1). \tag{3}$$

If $|D(x)| = |D(y)| = 2$ and $0 \in D(x) \cap D(y)$, we apply Boolean conjunction instead. All these decompositions seem reasonably strong because experimentation with McCormick envelopes [18] did not show better results.

3.2 Linearization of Domain Constraints

A critical class of constraint for linearization are the so called domain constraints. Under *domain constraint* for variable $x \in \mathbb{Z}$ we understand any of the following:

$$x \in S, \tag{4a}$$

$$\beta \;\leftrightarrow\; x \in S, \tag{4b}$$

where $S \subset \mathbb{Z}$ is a finite integer set and β a Boolean variable. (4a) is a static and (4b) is a reified domain constraint. In FlatZinc they are imposed by predicates `set_in(_reif)`.

This class of constraints generalizes some other non-linear constraints, such as comparisons with a constant: $x \neq a$ (static and reified, `int_ne(_reif)`), $x = a$ and $x \leq a$ (reified, `int_(eq/le)_reif`). Of the two latter, only reified versions are non-linear. W.l.o.g., FlatZinc doesn't consider other comparison operations as they can be reduced to "$\leq$" by variable substitution.

Moreover, comparisons between two variables $x, y \in \mathbb{Z}$ can be transformed to constraints (4a), (4b) by introducing a variable for their difference: $z = x - y$. Then, for example, $x \neq y$ is equivalent to $z \neq 0$.

Domain constraints (4a) and (4b) can be straightforwardly linearized using the unary encoding (2) by the following constraints (5a) and (5b), respectively:

$$1 = \textstyle\sum_{k \in S} b^x_k, \tag{5a}$$

$$\beta = \textstyle\sum_{k \in S} b^x_k. \tag{5b}$$

Domain Refinement for Integer Variables. The unary encoding (2) can introduce a lot of auxilliary variables b^x_k for x with a big domain $D(x) \subset \mathbb{Z}$. We propose a refined domain structure as follows. Let $x \in S$ be a constraint of the form (4a). Let the following list:

$$SL(S) = \arg\min \left\{ n \mid S = \mathbb{Z} \cap \bigcup_{i=1}^{n} [l_i, u_i],\ l_i, u_i \in S,\ i = \overline{1, n} \right\} \tag{6}$$

be the smallest list of integer-bounded intervals covering S and including no other integer values. Then, $x \in S$ is equivalent to the following system:

$$\textstyle\sum_i l_i \tilde{b}_i \le x \le \sum_i u_i \tilde{b}_i, \tag{7a}$$

$$\textstyle\sum_i \tilde{b}_i = 1, \tag{7b}$$

$$\tilde{b}_i \in \{0,1\}, \qquad i = \overline{1, |SL(S)|}. \tag{7c}$$

System (7) generalizes the unary encoding (2). Note that when the full unary encoding `eq_encode(x)` is already introduced for a given variable x, the MiniZinc transformation ensures that it is used for the domain constraints instead of the above. This prevents both systems (2) and (7) from being present in the model.

System (7) can be seen as a disjunction of 1D polyhedra. However we are not aware of any previous results in line with the unified domain refinement introduced below.

Unified Domain Refinement. We propose a single domain refinement which can be used to decompose all the domain constraints on a given variable, as well as those on dependent variables.

Example 2 (continued from Example 1). For the model of Fig. 1, consider the following list of intervals:

$$\overline{SL} = ([0,4],[5,6],[7,10]),$$

and the corresponding system (7). Then we can linearize the model by imposing the following equivalences:

$$\beta_1 = \tilde{b}_1, \tag{8a}$$

$$\beta_2 = \tilde{b}_3. \tag{8b}$$

The solution `beta1 = beta2 = 0.75; x == 5.5;` of Example 1 is no longer feasible, in the linear relaxation, simply by (7b). □

W.l.o.g., for a given variable x we have exactly one static domain constraint (4a) (with $S = D(x)$) and possibly several reified constraints (4b).

Definition 1. *Given an integer variable x and all its domain constraints*

$$x \in D(x), \tag{9a}$$

$$\beta_j \;\leftrightarrow\; x \in S_j, \qquad j \in J_x, \tag{9b}$$

define the unified domain refinement $\overline{SL}_x$ *as the list of the isolated intervals of the set*

$$\mathcal{S}_x = SL(D(x)) \cap \bigcap_{j \in J_x} \left(SL(S_j) \cup SL(D(x) \setminus\ S_j) \right). \tag{10}$$

Note that unary encoding (2) represents a special case of domain refinement, namely it is equivalent to system (7) based on the degenerate interval list $([l_k, u_k] \mid l_k = u_k = k,\ k \in D(x))$.

Theorem 1. *For an integer variable x, system* (7) *based on the interval list $\overline{SL}_x$ correctly linearizes the static constraint* (9a)*. For each $j \in J_x$, the reified constraint* (9b) *is correctly linearized by the following equation:*

$$\beta_j = \sum \left\{ \tilde{b}_i \mid l_i, u_i \in S_j,\ i \in \left\{1, \ldots, \left|\overline{SL}_x\right|\right\} \right\}. \tag{11}$$

Proof. Note that $\mathcal{S}_x \cap \mathbb{Z} = D(x)$, which proves correctness for (9a). The sublist $([l_i, u_i] \mid l_i, u_i \in S_j)$ of $\overline{SL}_x$ covers all elements of $S_j \cap D(x)$, proving (11). □

Theorem 2. *For an integer variable x with domain constraints* (9)*, the continuous relaxation of the decompositions* (7)*,* (11) *based on unified domain refinement $\overline{SL}_x$ as well as that of unary encoding* (2)*,* (5b) *are equally strong in terms of the high-level variables (x and β_j, $j \in J_x$).*

Proof. Given a continuous solution $\left(x, \beta_j|_{j \in J_x}, b^x_k|_{k \in D(x)}\right)$ of unary encoding, it is easy to see that $\left(\tilde{b}_i = \sum_{k=l_i}^{u_i} b^x_k\right)\Big|_{i=1}^{|\overline{SL}_x|}$ fulfills (7) and (11).

Vice versa, given a continuous solution $\left(x, \beta_j|_{j \in J_x}, \tilde{b}_i|_{i=1}^{|\overline{SL}_x|}\right)$ of (7), (11), set

$$r = \frac{x - \sum_i l_i \tilde{b}_i}{\sum_i u_i \tilde{b}_i - \sum_i l_i \tilde{b}_i} \in [0, 1].$$

For each $i \in \left\{1, \ldots, \left|\overline{SL}_x\right|\right\}$, if $l_i = u_i$ then set $b^x_{l_i} = \tilde{b}_i$, otherwise select $(b^x_k)|_{k=l_i}^{u_i}$ so that $\tilde{b}_i = \sum_{k=l_i}^{u_i} b^x_k$ (which provides (2a) and (5b)) and

$$\left(\sum_{k=l_i}^{u_i} k \frac{b^x_k}{\tilde{b}_i} - l_i \right) \Bigg/ (u_i - l_i) = r,$$

which is in general non-unique. This fulfills (2b):

$$\sum_{k \in D(x)} k b^x_k = \sum_{i=1}^{|\overline{SL}_x|} \sum_{k=l_i}^{u_i} k b^x_k = \sum_{i=1}^{|\overline{SL}_x|} \tilde{b}_i (r(u_i - l_i) + l_i) = x. \qquad \square$$

We see that unary encoding can have non-unique equivalents for a solution of unified refinement, leading to symmetries.

Dependent variables. When there is a set of variables $\{x_1, \ldots, x_n\}$ that are pairwise linearly dependent (i.e. $\forall 1 \le i < j \le n \exists a_{ij} b_{ij}$ s.t.$x_i = a_{ij} x_j + b_{ij}$), if at least one of them has a unary encoding generated by a specialized global, it can be re-used for the domain constraints of all $\{x_i\}$. Otherwise, all the domain constraints on $\{x_i\}$ can be projected onto just one of them, using a single unified domain refinement.

The unification procedure was implemented as a post-processing step in the MiniZinc compiler v2.0.10 but still controllable from the redefinition library. It looks for linearly dependent variables in several ways, for example if two variables are initialized by linear expressions whose non-constant parts are multiples of each other. This occurs for various auxiliary variables introduced in reformulations.

3.3 Global Constraint Decompositions

The MiniZinc distribution defines default decompositions for over 100 global constraints. The reformulations described in 3.2 above ensure that most of these default decompositions can be directly re-used for MIP, producing tight linear reformulations of the global constraints, where duplication of auxiliary variables has been automatically minimised.

For a few constraints where default decomposition is not MIP-efficient, we have implemented tailored MIP formulations, listed in the directory `share/minizinc/linear` of the MiniZinc distribution.

`alldifferent`, `inverse`, `alldifferent_except_0`: We have already seen how `alldifferent` is linearised using the unary encoding. One can linearise `inverse` similarly. The constraint `alldifferent_except_0` is a simple variation of `alldifferent` and much more pleasing than the constraint programming decomposition:

```
predicate alldifferent_except_0(array [Set1] of var Set2: x)  =
    forall (j in Set2 diff {0})( sum(i in Set1)(x[i]==j) <= 1 );
```

`element`, `table`: We have already seen how `element` is linearised using the unary encoding. The table constraint `table`$([x_1, \ldots, x_n],\ T)$ is encoded by defining auxiliary 01 variables $\lambda_i, 1 \leq i \leq m$ for each of the m rows in the table and then equating $x_j = \sum_{i=1}^{m} \lambda_i T_{ij}$. This is a direct extension of the `element` encoding, minus the index element.

`cumulative`: The global `cumulative` constraint, limiting the total amount of a renewable resource available to all tasks at any moment of time, is frequent in scheduling problems [25]. It can be used to express `alldifferent`, as in the `ghoulomb.mzn` benchmark, and as a redundant constraint in packing problems [26].

Two forms of reasoning used in the cumulative constraints are reasoning about the ordering of tasks ("task decomposition") [25], and reasoning about the tasks running at each time slot ("time decomposition") [12].

Transforming the cumulative constraint for MIP can be costly in terms of both the number of variables and constraints. While the number of variables resulting from the task decomposition is proportional to the number of tasks squared, the time decomposition ultimately requires a variable for each task indicating its relation to each time slot, which requires a number of variables proportional to the product of tasks and time slots.

The time decomposition of `cumulative` is currently the default in MiniZinc, and thus was solely used by the previous linearization library. We found that when the product of the number of time slots and the number of tasks exceeds a certain parameter (chosen as 2000), it is advantageous to use the task decomposition of `cumulative` and not the time decomposition.

`circuit` **and** `subcircuit`**:** The global constraints `circuit` and `subcircuit` take an argument vector `x`, where `x[i]` denotes the successor of node `i` (or just `i` if it is not included in the subcircuit). They ensure that there are no separate cycles and each node is in exactly (for `subcircuit`, in at most) one loop.

The previous linearization library had no special translation for them, resulting in the usage of standard decompositions. As an example, for subcircuit they involved ordering constraints of the type

(⟨*ordering condition*⟩) `-> order[x[i]] = order[i] + 1`

where auxiliary variable `order[i]` is the order of node `i` in the subcircuit, starting from the least-index node. The order of excluded nodes is not constrained. Expression `order[x[i]]` is a variable subscript (flattened as predicate `array_var_int_element`) and hard to linearize efficiently.

These globals are now encoded as variants of the Miller-Tucker-Zemlin formulation [20]. Interestingly, in an experiment with the lifted MTZ cuts of Desrochers and Laporte [9], we observed inferior behaviour when we tested them using IBM ILOG CPLEX 12.6.1 [14].

`regular`**:** Probably the most difficult global constraint for the previous linearization library is `regular`. It requires that the sequence of values in the control vector `x` satisfies a deterministic finite automation defined by the acceptable states vector `a` and a transition function `d` mapping the current state and the control value into the next state: `a[i+1] = d[a[i], x[i]]`. The default decomposition just uses the latter prescription directly, resulting in a series of `element`'s.

Specialized propagation algorithms for `regular`, cf. [8], construct the graph of achievable/feasible states for each step, called *layered graph*. Its nodes correspond to the unary encodings of the state variables `a[i]` for each step `i`: node (i, k) means `a[i]==k`, and arcs denote the transitions between the nodes of the consecutive steps (layers). A network-flow approach in [8] uses such a graph, implemented by an external procedure, and formulates the network-flow constraints in a MIP-typical way, namely with binary variables $\lambda_{(i,k_1),(i+1,k_2)} \in \{0, 1\}$ for the flow on each arc $((i, k_1), (i + 1, k_2))$.

We implemented this reformulation in MiniZinc, iteratively tightening the domains of the state variables `a[i]` and introducing the above-mentioned arc flow variables.

4 Experiment

To validate the MIP reformulations described above, we tested them on leading commercial and free MIP solvers, and compared them with the best solvers based on results from the MiniZinc Challenge.

As test instances we used 400 instances from MiniZinc Challenges 2012–2015. Naturally these instances are advantageous for the solvers proven on the very same test set!

The MIP solvers we tested were:

– commercial solver Gurobi 6.5.1 [11],
– commercial solver IBM ILOG CPLEX 12.6.3 [14],
– free solver COIN-OR Branch-and-Cut (CBC) 2.9.8 [17].

We tested the MIP solvers each under three configurations: default (with all linearization approaches), "no DR" (without domain refinement, Section 3.2), and "old" (with the old linearization library from MiniZinc 1.6 however supplemented with MIP-tailored globals `alldifferent`, `table`, and `inverse`, Sections 2 and 3). The multi-pass compilation of models suggested in [16] was not considered as it currently fails on 15 instances.

The best solvers from the MiniZinc Challenge we used for comparison were:

– Opturion CPX [22], overall official winner of the Challenges 2013 and 2015,
– Chuffed [5, 6], not prize-eligible in the Challenge.

For these solvers we used the search strategy specified by the model.

All solvers were executed sequentially (1 thread) on an Intel i7-4771 CPU @ 3.50 GHz with a memory limit of 12 GB per process. MiniZinc 2.0.13[3] was used to flatten the models. The actual FlatZinc-to-solver interfaces are going to be released as part of the upcoming MiniZinc 2.1. Solving time was limited to 5 minutes total CPU time per method/instance. Flattening time was not limited.

In Table 1 we report the following data for each solver and configuration: the number of optimal (opt), feasible but not optimal (feas), satisfied (sat), proven infeasible (inf), not flattened (nofzn), failed (fail) (solver crashed or did not stop normally in 500s), and other cases (other) — where none of the previous results were achieved. i.e., the solver ran without finding feasible solutions and did not crash.

Table 1. Comparison of solvers and configurations

Solver+config	opt	feas	sat	inf	nofzn	fail	other
Gurobi	160	113	48	3	5	1	70
Gurobi, noDR	157	115	45	2	5	2	74
Gurobi, old	133	118	28	5	16	4	96
CPLEX	139	121	46	2	5	1	86
CPLEX, noDR	141	124	47	3	5	0	80
CPLEX, old	124	118	26	4	16	2	110
CBC	86	82	24	2	5	32	169
CBC, noDR	78	69	20	1	5	40	187
CBC, old	61	58	8	1	16	51	205
Chuffed	157	142	54	5	2	0	40
Opturion CPX	130	159	37	5	2	0	67

[3] minizinc.org

1. Note from Table 1 that sometimes the MiniZinc-to-FlatZinc compilation cannot produce a working model (columns `nofzn` and `fail`). For MIP, the main causes were the globals `cumulative`, `regular` and `table` involving variables with large domains, leading to huge decompositions and thus to big MIP models where MIP solvers run out of memory or just stop responding. The same happens to the CP solvers when they don't handle a global constraint directly and it has to be decomposed, in this case `cumulative` with variable durations and resource demands in 2 instances of `mznc2013/fjsp`.
2. The linear solvers successfully find and prove optimality - Gurobi beats both Chuffed and Opturion CPX in terms of number of optimal solutions.
3. The new domain refinement is definitely beneficial to Gurobi and more so for CBC, while strangely it is disadvantageous for CPLEX. We believe it may interfere with some presolve simplifications in CPLEX.

Even the free MIP solver CBC, with helpful support from its developers, now runs bug-free and gives results on nearly half the instances. Moreover, as revealed in Table 2 CBC sometimes succeeds where the Challenge solvers fail.

In Table 2 we present pairwise set difference analysis for the solvers' best configurations. For each comparison of two solvers/configurations, we report the following numbers: `Oopt` is the number of cases where only that solver proved optimality (and the other solver had at best feasibility); `Ofea` is the number of cases where only that solver found a feasible solution (and the other none); `Oinf` is the similar value for infeasible cases; `Bpri` and `Bdua` are the numbers of instances for each solver when it has found a better primal/dual bound if both had one, respectively.

Gurobi outperforms Chuffed on over 100 instances, and even CBC outperforms Chuffed on over 50 instances. The differences between the MIP solvers and the Challenge solvers are particularly evident on proofs of optimality and feasibility (`Oopt` and `Ofea`). In this comparison, CBC outperforms Chuffed on 30 instances, while CPX only outperforms Chuffed on 8 instances.

What emerges from these tests is that, now that the linearization of CP models has been improved, MIP shows complementary strengths in contrast with the Challenge solvers. Given a new MiniZinc model it makes sense to try eveluating it with different classes of solvers, including MIP.

The work that has been done ensuring the behaviour of CBC is sound can also pay off by enabling constraint programmers to have immediate access to a free MIP solver adding an additional weapon to the CP modeller's armoury.

5 Conclusion

The results show that MIP solvers are highly competitive with CP solvers on MiniZinc benchmarks, which are, for the most part, written with CP solvers in mind. This is good news since it validates the constraint programming view of modelling: that the model should be written in the highest level possible, and it should be up to tools to map this to a suitable form for the solver if

Table 2. Comparison of difference sets

Solver+config	Oopt	Ofea	Oinf	Bpri	Bdua
	Gurobi vs Chuffed				
Gurobi	52	22	0	41	-
Chuffed	49	45	2	23	-
	CPLEX(noDR) vs Chuffed				
CPLEX(noDR)	40	19	0	42	-
Chuffed	56	50	2	29	-
	CBC vs Chuffed				
CBC	21	9	0	25	-
Chuffed	92	112	3	24	-
	Gurobi best cfg vs CPLEX best cfg				
Gurobi	21	19	0	46	98
CPLEX(noDR)	3	14	0	33	31
	Chuffed vs Opturion CPX				
Chuffed	30	26	0	49	-
CPX	3	5	0	37	-

needed (of course the CP perspective is that the preferred mapping would be to a global propagator, but also MIP solvers might start providing global constraint handlers). This is also a challenge to CP since it illustrates that MIP solvers can be used out of the box to tackle problems that we often consider are suited to the CP solving technology. Of course there is a place for both CP and MIP solving technology, and one of the aims of a solver-independent modelling language is to avoid users committing early to the wrong technology. Better linearization makes MiniZinc a more attractive modelling language for the general OR community, which may then make them more aware of the CP view of modelling and solving.

There is plenty of scope for further improvement of automatic linearization of MiniZinc models. Issues that we plan to investigate are: better continuous relaxations of nonlinear expressions, avoiding symmetry creation in decompositions, providing a declarative interface to domain refinement to allow the user to control the process using annotations.

References

1. A list of constraint languages. http://www.csplib.org/Languages/, 2016.
2. N. Andrei. Introduction to GAMS technology. In *Nonlinear Optimization Applications Using the GAMS Technology*, volume 81 of *Springer Optimization and Its Applications*, pages 9–23. Springer US, 2013.
3. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

4. G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, 2015.
5. G. Chu. Improving combinatorial optimization - extended abstract. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China*, pages 3116–3120, 2013.
6. G. Chu. Constraint Programming solver Chuffed, 2016. https://github.com/geoffchu/chuffed, downloaded on 16 March 2016.
7. A. A. Cire, J. N. Hooker, and T. Yunes. Modeling with metaconstraints and semantic typing of variables. *INFORMS JoC*, 28(1):1–13, 2016.
8. M.-C. Côté, B. Gendron, and L.-M. Rousseau. Modeling the regular constraint with integer programming. In P. Van Hentenryck and L. Wolsey, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2007.
9. M. Desrochers and G. Laporte. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. *Operations Research Letters*, 10(1):27–36, 1991.
10. R. Fourer and D. M. Gay. Extending an Algebraic Modeling Language to support Constraint Programming. *INFORMS Journal on Computing*, 14(4):322–344, 2002.
11. Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual Version 6.5*. Houston, Texas: Gurobi Optimization, 2016.
12. J. R. Hardin, G. L. Nemhauser, and M. W. P. Savelsbergh. Strong valid inequalities for the resource-constrained scheduling problem with uniform resource requirements. *Discrete Optimization*, 5(1):19–35, 2008.
13. J. N. Hooker. *Integrated Methods for Optimization*, volume 170 of *International Series in Operations Research & Management Science*. Springer US, 2012.
14. IBM Software. IBM ILOG CPLEX optimizer. Data sheet, IBM Corporation, 2014.
15. Thorsten Koch. *Rapid Mathematical Prototyping*. PhD thesis, Technische Universität Berlin, 2004.
16. K. Leo and G. Tack. Multi-pass high-level presolving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.
17. J. Linderoth and T. Ralphs. Noncommercial software for Mixed-Integer Linear Programming. Technical report, Lehigh University, 2004.
18. G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I — convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976.
19. K. McKinnon and H. Williams. Constructing integer programming models by the predicate calculus. *Annals of Operations Research*, 21, 1989.
20. C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, October 1960.
21. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
22. Opturion Pty Ltd. Opturion CPX user's guide: version 1.0.2, 2013.
23. P. Refalo. Linear formulation of Constraint Programming models and hybrid solvers. In R. Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer Berlin Heidelberg, 2000.
24. D. Salvagnin. Detecting semantic groups in MIP models. *CPAIOR 2016 Proceedings*, 2016.

25. A. Schutt, Th. Feydy, P. J. Stuckey, and M. G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011.
26. A. Schutt, P. J. Stuckey, and A. R. Verden. Optimal carpet cutting. In J. Lee, editor, *Principles and Practice of Constraint Programming — CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin Heidelberg, 2011.
27. P.J. Stuckey, R. Becket, and J. Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.
28. P.J. Stuckey and G. Tack. MiniZinc with functions. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2013.