Updated Linearization Library for MiniZinc 2.0

Gleb Belov, Guido Tack, and Mark Wallace

Monash University, Caulfield Campus, Australia
{name.surname}@monash.edu

Abstract. MiniZinc is a solver-independent Constraint Programming (CP) modeling language. The solver interface works by translating a MiniZinc model into the simpler language FlatZinc. A specific solver can provide its own redefinitions of MiniZinc constraints.
Most CP models are highly non-linear. MiniZinc offers a linearization library to translate into Mixed-Integer Linear Programming (MILP) models. Modern MILP solvers accept linear and some non-linear constraints, such as logical ones.
This paper describes improvements to the redefinitions for MILP solvers, that we have introduced to enhance the performance of MILP on MiniZinc models. We discuss known and new linearization methods and effects of certain redundant constraints. The library was tested on the benchmarks of the MiniZinc Challenges 2012–2014, and on other interesting MiniZinc models.
Experiments show the new redefinitions improve MILP performance on the Challenge problems, and can enable the MiniZinc modeller to gain good performance from MILP without adapting the MiniZinc model.

Keywords: automatic reformulation, linear decomposition, MiniZinc Challenge

1 Introduction

MiniZinc is fast becoming the de facto standard modeling language for the constraints community. The MiniZinc front-end is now supported by some 20 solvers, including finite domain solvers, SAT solvers, Lazy Clause Generation (SMT) solvers, and even local search solvers [6]. Annual MiniZinc competitions [14] provide a basis for comparing solvers and exploring their strengths and weaknesses.

A number of modeling front-ends are available for MILP solvers, including GAMS [4] and AMPL [3]. MiniZinc can also be used as a modeling front-end for MILP solvers, but has not yet achieved recognition in the OR community. In the MiniZinc Challenge MILP solvers have had some success, for example on the bin-packing problem instances, but MILP does not appear competitive on most of the Challenge benchmarks.

Our vision is that MiniZinc becomes an accepted and even widely-used modeling language within the OR community, thus helping to narrow the divide between OR, CP and SAT researchers. To this purpose we seek to ensure that

MILP models, when formulated in MiniZinc, have at least similar performance to AMPL and GAMS. Beyond that, MiniZinc offers the advantage of MILP-compatible reformulation of CP models, and the efficiency of that seems a more challenging task.

As a step towards this goal we have begun to explore why MILP has not to date been competitive across the board in the MiniZinc Challenges. This may partly be on account of the class of problems used (for example problems involving non-linear constraints and many integer variables). However another reason for this may be that the MiniZinc flattening process has not generated the best MILP models.

In this paper we explore MiniZinc's flattening translation for models mapped to MILP solvers. We identify potential sources of inefficiency in the flattened model, and we explore alternative translations which may yield better flattened models.

The version of MiniZinc we use for testing is 2.0.5 [9]. Its default MILP library `share/minizinc/linear` contains linearizations of the basic constraints, such as multiplications and logical operations, as well as of some *global constraints* [5], namely `table_int`, `inverse` and `all_different_int`.

We propose a new library, located in folder `share/minizinc/linear_new` of the MiniZinc 2.0.5 distribution. The library contains tighter formulations of basic constraints, as well as specialized decompositions of globals. Some of the decompositions are controlled by parameters. The new library was submitted to MiniZinc Challenge 2015.

Below we discuss the new linearization methods and provide numerical comparisons, also to some CP solvers.

2 Linear Decompositions

A specialized translation library can be provided as a set of files, typically in a separate subdirectory of `share/minizinc/`. The name of the subdirectory is passed after the `-G` flag of `mzn2fzn`. Such a library does not have to contain all possible constraints; anything not re-defined is taken from the standard library contained in `share/minizinc/std`.

In particular, files `redefinitions*.mzn` re-define the basic constraints, such as logical ones, min/max, `element`, etc. Most global constraints are specified in dedicated files, such as `lex_less.mzn`, because a model includes them only if necessary.

2.1 Unary Encoding of Integers

Probably the most frequent non-linear constraint in our benchmarks is a disequality between an integer variable and another one or a constant value. Here we use the technique proposed by Refalo [11] for domain constraints, however the domain constraints themselves we handle differently, see the next subsection.

Disequality of two integer variables can be of the static form (predicate `int_ne` in `redefinitions.mzn`)

$$x \neq y \tag{1}$$

but it also crops up in reified equality (`int_eq_reif`)

$$x = y \quad \Leftrightarrow \quad b = true. \tag{2}$$

If $b \equiv true$ this is a linear constraint.

There are two alternative transformations for handling disequality involving a variable x. The first is a disjunction between two inequations (y may be a constant or a variable):

$$x \geq y + 1 \vee y \geq x + 1 \tag{3}$$

which must in turn be transformed by introducing a boolean variable and two "big-M"s.

The second alternative is to introduce a boolean variable for each value in the domain of x [11]. If b_k^x is the boolean variable corresponding to the value k in the domain of x, then the disequality $x \neq k$ has a simple transformation as

$$b_k^x = 0. \tag{4}$$

Disequality between two variables, x and y can be reduced to the form $x - y \neq 0$, and handled as above by introducing boolean variables for the domain of $x - y$.

Nonetheless, there are many further constraints which are best transformed using these boolean variables, including `all_different`, `element`, multiplication of variables, and some others.

To achieve a tight MILP model without duplicate variables and constraints, it is essential that every time a constraint on a same variable is transformed by introducing these booleans, the very same booleans are used in the transformation. This is achieved automatically through MiniZinc's functions [15].

To introduce the boolean variables, as proposed in [15], we use function `eq_encode(var int: x)` (there, called `int2array`), defined in `domain_encodings.mzn`, returning an array of 0–1 variables and imposing linear constraints such that `eq_encode(x)[i]=1 <-> x=i`.

Now every time this function is invoked on a variable x, MiniZinc's common subexpression elimination ensures that the same booleans are reused, even if the function is embedded in a predicate or another function.

2.2 Domain Constraints for Integer Variables

Domain constraints restrict an integer variable to accept values in a given set (predicate `set_in`), also in a reified form (`set_in_reif`).

For the static version `set_in` we propose the following reformulation of $x \in S$. Let

$$SL = \{[l_i, u_i] \mid i \in I_S\}$$

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

$$\begin{aligned} \textstyle\sum_i l_i f_i \le x \le \textstyle\sum_i u_i f_i, \\ \textstyle\sum_i f_i = 1, \\ f_i \in \{0,1\}, \qquad \forall i. \end{aligned}$$

For the reified version `set_in_reif`,

$$x \in S \qquad \Leftrightarrow \qquad b = true, \tag{5}$$

let us denote by $lb(x)$ and $ub(x)$ the finite lower and upper bounds on x, respectively. Then (5) is equivalent to the following system:

$$\begin{aligned} lb(x) + \textstyle\sum_i f_i(l_i - lb(x)) \le x \le ub(x) + \textstyle\sum_i f_i(u_i - ub(x)), \\ \textstyle\sum_i f_i = \texttt{bool2int}(b), \\ f_i \in \{0,1\}, \qquad \forall i. \end{aligned}$$

These reformulations generalize the unary encoding for the domain constraint proposed in [11]. Note that when the full unary encoding `eq_encode(x)` is introduced for x somewhere else, we should use it for the domain constraints instead of the above. Current capabilities of MiniZinc do not allow such a query. This is the reason why the above suggestion does not work well for `set_in_reif`, solving only 1 instance optimally in `2014/solbat` instead of 3. So we currently use the full unary encoding for `set_in_reif`.

2.3 Unary Encoding for Multiplication of Integer Variables

Predicate `int_times` constraining $z = xy$ was previously implemented as `element`$((xy_1, \ldots, xy_n), y)$, where $\{y_1, \ldots, y_n\}$ is the domain of y. This method would also work with x real-valued.

In the cases of a small (chosen as 4..20) product domain size $|\operatorname{dom}(x)| \times |\operatorname{dom}(y)|$, experiments proved that it is advantageous to use the following alternative encoding:

$$z = \textstyle\sum_{i,j} i \times j \times b_{ij}^{xy}, \qquad \text{where} \quad b_{ij}^{xy} = 1 \leftrightarrow (b_i^x = 1 \wedge b_j^y = 1). \tag{6}$$

If $|\operatorname{dom}(x)| = |\operatorname{dom}(y)| = 2$ and $0 \in \operatorname{dom}(x) \cap \operatorname{dom}(y)$, we apply boolean conjunction instead.

2.4 Global Constraint `cumulative`

Global constraint `cumulative`, limiting the total amount of a renewable resource available to all tasks at any moment of time, is frequent in scheduling problems [12]. It can be used to express `alldifferent`, as in the `ghoulomb.mzn` benchmark, and as a redundant constraint in packing problems [13].

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

Transforming the cumulative constraint for MILP 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 squared number of tasks, 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.

A comparison is given in Table 1. For each instance using `cumulative` we report 2 configurations, with flexible cumulative and time decomposition only. For each configuration we report 5 values: t_{flt} is flattening time (0 if failed due to memory overflow), obj and bnd are the best found objective value and dual bound, t_{slv} is the solution time, and sta is the status: `O, F, I, ?,` ☠ for optimal, feasible, infeasible, unknown, failure (not flattened/memory overflow), respectively. Feasibly solved satisfiability problems are reported as optimal with objective value 0. Experimental setting was the same as in Section 3, in particular flattening time was not limited and solving time was limited by 5 minutes total CPU time per method/instance.

2.5 Global Constraints `circuit` and `subcircuit`

Global constraints `circuit` and `subcircuit` take an argument vector `x`, where `x[i]` denotes the successor of node `i` (or just `i` if 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 `array_var_int_element` and hard to linearize efficiently.

Now these globals are encoded as variants of the Miller-Tucker-Zemlin formulation [10], changing, e.g., the above condition to a series of constraints

```
order[i] - order[j] + n * bool2int( x[i]==j /\ i!=lastin ) <= n-1,
```

where `lastin` is the last node in the subcircuit before the least-index node.

Table 1. Effect of the flexible global `cumulative`

Problem	Instance	flexible `cumulative`					time decomp				
		t_{flt}	obj	bnd	t_{slv}	sta	t_{flt}	obj	bnd	t_{slv}	sta
cc_base	_rnd_test.05	1.2	1206	1050.0	300.0	F	377.4	–	–	–	💀
cc_base	_rnd_test.10	6.5	–	3551.0	300.1	?	0.0	–	–	–	💀
cc_base	_rnd_test.14	2.4	2544	1640.0	300.1	F	0.0	–	–	–	💀
cc_base	_rnd_test.16	1.2	1394	1075.0	300.0	F	373.3	–	–	–	💀
cc_base	_rnd_test.17	5.1	–	1862.0	300.1	?	0.0	–	–	–	💀
mspsp	easy_01	1.4	26	26.0	1.6	O	8.1	26	26.0	5.4	O
mspsp	hard_01	1.3	35	33.0	300.0	F	5.2	35	32.4	300.3	F
mspsp	hard_03	2.1	30	30.0	253.1	O	10.5	30	28.0	300.1	F
mspsp	medium_02	0.5	15	15.0	0.4	O	3.8	15	15.0	2.0	O
mspsp	medium_03	1.4	26	26.0	1.6	O	9.1	26	26.0	7.0	O
mspsp	medium_05	1.1	18	18.0	3.6	O	6.0	18	18.0	8.0	O
cargo	04_1s_626	0.7	714	714.0	20.6	O	0.0	–	–	–	💀
cargo	05_1s_954	0.8	2883	2883.0	17.7	O	0.0	–	–	–	💀
cargo	07_1s_133	1.2	–	328.0	300.0	?	0.0	–	–	–	💀
cargo	_222f_3475	1.6	28869	19149.7	300.0	F	0.0	–	–	–	💀
cargo	_15966f_2060	2.0	–	4395.6	300.1	?	0.0	–	–	–	💀
fjsp	easy01	0.8	253	253.0	1.6	O	0.0	–	–	–	💀
fjsp	easy02	2.4	79	11.0	300.1	F	132.5	–	0.0	301.9	?
fjsp	hard19	2.3	–	4037.0	300.1	?	0.0	–	–	–	💀
fjsp	med04	0.0	–	–	–	💀	0.0	–	–	–	💀
fjsp	med10	0.5	942	741.3	300.0	F	0.0	–	–	–	💀
ghoulomb	3-11-29	0.0	–	–	–	💀	0.0	–	–	–	💀
ghoulomb	3-9-16	10.4	–	15.0	300.2	?	87.5	–	24.0	300.6	?
ghoulomb	4-9-10	2.1	53	23.0	300.1	F	19.3	–	27.0	300.1	?
ghoulomb	4-9-20	26.3	–	13.0	300.6	?	216.2	–	9.0	303.2	?
ghoulomb	5-7-22	40.4	–	19.9	300.8	?	329.8	–	–	–	💀
rcpsp	11	0.7	78	59.0	300.0	F	12.8	78	64.0	300.1	F
rcpsp	12	0.7	38	38.0	19.4	O	3.1	38	38.0	22.4	O
rcpsp	13	0.2	78	69.0	300.0	F	9.2	78	78.0	99.1	O
rcpsp	14	1.7	138	95.2	300.0	F	27.0	194	94.0	300.3	F
rcpsp	15	10.3	318	166.0	300.2	F	125.7	–	160.0	300.9	?
rcmsp	easy_3	1.5	1920	1919.0	19.2	O	7.2	1920	1920.0	30.6	O
rcmsp	easy_5	0.8	862	862.0	26.7	O	6.6	862	862.0	38.0	O
rcmsp	hard_2	6.3	–	1537.0	300.2	?	90.0	–	1609.6	300.8	?
rcmsp	hard_5	0.7	795	795.0	138.2	O	3.3	795	795.0	77.0	O
rcmsp	medium_2	0.4	516	516.0	7.2	O	1.9	516	516.0	7.2	O
openshop	gp10-4	1.8	1087	1000.0	300.0	F	0.0	–	–	–	💀
openshop	j7per10-1	0.5	1000	1000.0	30.1	O	0.0	–	–	–	💀
openshop	j8per0-2	0.8	1059	1000.0	300.0	F	0.0	–	–	–	💀
openshop	tai20-4	16.4	–	1248.0	300.4	?	0.0	–	–	–	💀
openshop	tai20-6	16.9	–	1204.0	300.3	?	0.0	–	–	–	💀
rect_packing	rpp18_true	0.7	0	0.0	1.6	O	7.2	0	0.0	18.0	O
rect_packing	rpp21_false	1.0	0	0.0	5.2	O	10.3	0	0.0	38.0	O
rect_packing	rpp22_false	1.2	0	0.0	3.5	O	11.7	0	0.0	45.0	O
rect_packing	rpp23_false	1.1	0	0.0	7.5	O	14.4	0	0.0	69.8	O
rect_packing	rpp26_false	1.0	0	0.0	69.0	O	24.4	–	0.0	300.4	?
smelt	smelt_11	10.8	–	293.5	300.2	?	0.0	–	–	–	💀
smelt	smelt_2	0.1	69	69.0	0.3	O	45.6	5069	36.0	300.5	F
smelt	smelt_3	0.5	1117	1092.0	300.0	F	153.2	–	20.0	301.3	?
smelt	smelt_4	0.1	1043	1043.0	0.6	O	95.0	–	52.8	315.7	?
smelt	smelt_5	1.1	1168	1168.0	43.0	O	226.5	–	21.0	302.1	?

Table 2. Effect of the new globals `circuit` and `subcircuit`

Problem	Instance	new `(sub)circuit`					old				
		t_{flt}	obj	bnd	t_{slv}	sta	t_{flt}	obj	bnd	t_{slv}	sta
tpp	tpp_3_3_30_1	0.1	190	190.0	1.3	O	0.1	190	190.0	1.7	O
tpp	tpp_3_5_20_1	0.2	127	127.0	4.7	O	0.2	127	127.0	12.5	O
tpp	tpp_5_3_20_1	0.2	141	141.0	6.7	O	0.2	141	141.0	11.0	O
tpp	tpp_5_5_20_1	0.5	115	115.0	18.4	O	0.4	161	99.4	300.0	F
tpp	tpp_7_3_20_1	0.3	125	125.0	10.3	O	0.4	125	125.0	32.1	O
tpp	tpp_7_5_20_1	0.8	105	105.0	152.9	O	0.8	–	77.8	300.0	?
tpp	tpp_7_5_30_1	1.0	197	119.8	300.0	F	0.9	–	108.9	300.1	?
mario	mario_easy_2	0.1	628	628.0	0.1	O	0.1	628	628.0	1.3	O
mario	mario_easy_4	0.1	545	545.0	0.0	O	0.1	545	545.0	0.7	O
mario	mario_n_medium_2	0.5	1053	1053.0	0.2	O	0.5	1053	1053.0	18.8	O
mario	mario_n_medium_4	0.5	832	832.0	0.2	O	0.5	832	832.0	16.5	O
mario	mario_t_hard_1	6.0	4783	4783.0	104.5	O	4.0	–	4783.0	300.2	?
mario	mario_easy_5	0.1	445	445.0	0.4	O	0.1	445	445.0	1.6	O
mario	mario_n_medium_3	0.6	943	943.0	0.2	O	0.4	943	943.0	3.0	O
mario	mario_n_medium_5	0.6	771	771.0	0.5	O	0.4	771	771.0	26.1	O
mario	mario_t_hard_2	6.1	4850	4850.0	2.6	O	4.0	–	4850.0	300.2	?
mario	mario_t_hard_5	6.2	4482	4482.0	32.8	O	4.0	–	4482.0	300.2	?
vrp-s2-v2-c7_vrp-v2-c7_det		0.2	160	160.0	133.1	O	0.3	160	76.0	300.0	F
vrp-s3-v2-c6_vrp-v2-c6_det		0.2	194	194.0	237.6	O	0.3	194	122.0	300.0	F
vrp-s3-v2-c7_vrp-v2-c7_det		0.2	230	102.0	300.0	F	0.4	230	109.0	300.0	F
vrp-s4-v2-c6_vrp-v2-c6_det		0.3	221	221.0	290.8	O	0.5	221	142.1	300.0	F
vrp-s5-v2-c8_vrp-v2-c8_det		0.5	364	114.0	300.0	F	0.9	362	133.0	300.0	F

A numerical comparison is presented in Table 2. The stochastic VRP employs `circuit`, the other 3 problems `subcircuit`.

2.6 Global Constraint `regular`

Probably the most difficult global constraint for the current linearization library is `regular`. It requires that the sequence of values in the control vector `x` satisfies a deterministic finite automation defined by acceptable states and a transition function mapping the current state and the control value into the next state. The default decomposition uses a series of `element`s.

Specialized propagation algorithms for `regular`, cf. [7], construct the graph of achievable/feasible states for each step, called *layered graph*. A network-flow approach [7] uses such a graph, implemented by an external procedure, and formulates the network-flow constraints in a MILP-typical way, namely with binary variables $\lambda_{ij} \in \{0, 1\}$ for the flow on each arc ij.

We implemented this reformulation in MiniZinc. The layered graph is constructed by the following simplified procedure. Knowing the domain of the state variable a_i for automation step i and the transition function, we can find out the achievable states for step $i + 1$. Then this procedure is repeated backwards, starting from the acceptable states for the last step, and a third time forward.

Results are given in Table 3. The number of solved `nonogram`s grew from 2 to 10. For 4 of the `ttppv` instances, dual bounds were improved. For `pentominoes`,

Table 3. Effect of the new global `regular`

Problem	Instance	new `regular`					old				
		t_{flt}	obj	bnd	t_{slv}	sta	t_{flt}	obj	bnd	t_{slv}	sta
non	non_awful_3	9.5	0	0.0	2.3	O	10.4	–	0.0	300.5	?
non	non_awful_5	10.2	0	0.0	2.4	O	10.9	–	0.0	300.6	?
non	non_fast_11	12.7	0	0.0	3.5	O	12.9	–	0.0	300.7	?
non	non_fast_4	7.2	0	0.0	1.2	O	8.1	–	0.0	300.4	?
non	non_fast_8	13.3	0	0.0	3.1	O	12.0	–	0.0	300.7	?
non	dom_06	0.2	0	0.0	0.1	O	0.2	0	0.0	1.8	O
non	dom_08	0.4	0	0.0	0.1	O	0.3	0	0.0	84.0	O
non	dom_10	0.6	0	0.0	0.2	O	0.6	–	0.0	300.0	?
non	dom_12	0.9	0	0.0	0.3	O	0.8	–	0.0	300.1	?
non	dom_14	1.3	0	0.0	0.8	O	1.1	–	0.0	300.1	?
pentominoes-int	02	15.6	–	0.0	300.5	?	50.7	–	–	–	☠
pentominoes-int	04	0.0	–	–	–	☠	0.0	–	–	–	☠
pentominoes-int	05	19.9	–	0.0	300.6	?	62.3	–	–	–	☠
pentominoes-int	06	21.0	–	0.0	300.6	?	0.0	–	–	–	☠
pentominoes-int	07	24.2	–	0.0	300.6	?	0.0	–	–	–	☠
handball	handball11	2.2	–	0.0	300.1	?	3.2	–	0.0	300.2	?
handball	handball12	2.3	–	0.0	300.1	?	3.2	–	0.0	300.2	?
handball	handball13	2.3	–	0.0	300.1	?	3.1	–	0.0	300.1	?
handball	handball14	2.6	–	0.0	300.1	?	3.1	–	0.0	300.1	?
handball	handball19	2.7	–	0.0	300.1	?	3.1	–	0.0	300.1	?
ttppv	circ14bnonbal	3.4	–	35.0	300.2	?	3.3	–	17.0	300.2	?
ttppv	circ20gnonbal	11.6	–	0.0	300.6	?	11.0	–	0.0	300.6	?
ttppv	circ8bbal	0.5	88	47.8	300.0	F	0.5	88	43.8	300.0	F
ttppv	circ8cbal	0.5	84	48.7	300.0	F	0.5	86	45.7	300.0	F
ttppv	circ8ebal	0.5	80	50.1	300.0	F	0.5	82	46.0	300.0	F

2 more instances could be flattened but still none solved. Overall, multi-step automation seems to be hard for MILP. Such cases can probably be re-modeled more efficiently.

2.7 Stronger Formulations and Reduced Symmetries

Many boolean expressions have a naive translation into MILP that was previously used. For example consider the translation of a reified disjunction of booleans, `array_bool_or`,

$$b \leftrightarrow \bigvee_{i:1..n} a_i, \tag{7}$$

which was done by the following two linear constraints:

$$\begin{aligned} b &\leq \textstyle\sum_{i:1..n} a_i, \\ b * n &\geq \textstyle\sum_{i:1..n} a_i. \end{aligned}$$

Naturally a much tighter linearization, containing only facet-defining inequalities, is the following:

$$\begin{aligned} b &\leq \textstyle\sum_{i:1..n} a_i, \\ b &\geq a_i, \qquad\qquad \forall i. \end{aligned}$$

A similar tightening was introduced for reified conjunction of booleans, `array_bool_and`. Further tightened predicates include `int_abs`, `int_min/max`, `array_int_minimum/maximum`, `array(_var)_int_element` and their `float_` counterparts, `array_bool_xor`, `int_div`, `aux_int_eq_iff_1` (the "big-M" specialization of `int_eq_reif`), all defined in `redefinitions.mzn`.

2.8 Changes in the MiniZinc Compiler

Several changes have been introduced in the actual MiniZinc compiler, already in release 2.0.2, which improve model translation in general and especially for the case of translation to MILP. For example, nested `element` constraints are now simplified where possible. This is not very important for constraint propagation in CP, but essential for the strength of continuous LP relaxation.

3 Overall Experiment

The tests were run on an Intel i7-4771 CPU @ 3.50 GHz with a memory limit of 8 GB per process. MiniZinc 2.0.5 was used to flatten the models and IBM ILOG CPLEX 12.6.1 [8] as the MILP solver, executed sequentially (1 thread). We compared the results to two CP solvers, Google OR-Tools [2], winner of the MiniZinc Challenge 2014, and Chuffed [1], both with the fixed search strategy and sequential execution.

Again, solving time was limited by 5 minutes total CPU time per method/instance. Flattening time was not limited.

3.1 Comparison of Solvers and Configurations

We used the 300 instances of the 59 problems of the MiniZinc Challenges 2012–2014 [14]. We considered the following linearization configurations: (1.) default linearization library in MiniZinc 2.0.5; (2.) the same library with the new globals described in Section 2; (3.) the new library. Configurations (4.) and (5.) are Google OR-Tools and Chuffed, respectively.

Table 4 presents, for each problem and solver/setting, the number of instances solved optimally (O), only feasibly (F), proven infeasible (I), or not flattened (N). Flattened but unsolved instances are not reported. Feasibly solved satisfiability problems are reported as optimal.

3.2 Effects of Redundant `cumulative` Constraints

Exploring other avenues for improving MILP performance, we checked if it would be useful to drop redundant constraints. We tested this on the redundant cumulative constraints in a set of six problem classes.

Table 5 shows effects of removing the redundant `cumulative` constraints from the six problems that have it. There was a 100s time limit per instance.

Moreover, Table 6 shows the best objective value and lower bound for each instance of `carpet-cutting`, and also the best objective value from the 2012 Challenge (there, with a 900s time limit).

Experiments suggest removing redundancies is not always the best translation for MILP. Probably we can employ the "user cuts" mechanism of modern MIP solvers to introduce redundant constraints.

4 Conclusions

The results for MILP solvers in previous MiniZinc challenges have been patchy: some instances have failed to flatten, some have suffered from memory overflows during solving, some have yielded no feasible solutions, but for a few instances MILP has produced the best result.

In this paper we have looked into the reasons for these behaviours and we have improved several features of MiniZinc's flattening for MILP solvers. In particular we improved the redefinitions of basic constraints for MILP, we improved the translation of several global constraints and we improved MiniZinc's compilation of nested element constraints.

Our improvement in the translation of the global constraints `cumulative`, `circuit`, `subcircuit` and `regular` dramatically reduced the number of instances that could not be either flattened or solved by MILP under the previous treatment. Still, some constraints remain intrinsically hard for MILP, such as finite-domain automation with many steps, although they are not easy for CP either.

Also our improvements in the redefinitions and compilation enabled us, with the same (new) translation of the global constraints, to reduce the number of

Table 4. Comparison of linearization configurations and with CP solvers on the instances of the Challenges 2012–2014

	LIN Old				OldWGlb				LIN New				OR-Tools				Chuffed			
	O	F	I	N	O	F	I	N	O	F	I	N	O	F	I	N	O	F	I	N
2012																				
amaze	4	0	0	0	4	0	0	0	1	1	0	0	4	0	0	0	5	1	0	0
amaze2	1	0	0	0	1	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0
carpet-cutting	0	0	0	5	0	3	0	0	0	3	0	0	0	5	0	0	0	4	0	0
fast-food	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0	0	3	2	0	0
filters	2	3	0	0	2	3	0	0	2	3	0	0	5	0	0	0	4	1	0	0
league	4	1	0	0	4	1	0	0	5	1	0	0	0	6	0	0	2	4	0	0
mspsp	0	0	0	6	5	1	0	0	5	1	0	0	5	1	0	0	6	0	0	0
nonogram	0	0	0	2	5	0	0	0	5	0	0	0	3	0	0	0	5	0	0	0
parity-learning	1	0	0	0	1	0	0	0	3	0	0	0	7	0	0	0	0	0	0	0
pattern-set-mining-k1	1	1	0	0	1	1	0	0	0	2	0	0	2	0	0	0	0	2	0	0
pattern-set-mining-k2	1	2	0	0	1	2	0	0	1	2	0	0	1	2	0	0	1	2	0	0
project-planning	1	4	0	0	3	3	0	0	1	4	0	0	0	6	0	0	4	2	0	0
radiation	2	3	0	0	2	3	0	0	2	3	0	0	1	0	0	0	3	0	0	0
ship-schedule	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0	0
solbat	1	0	0	0	1	0	0	0	3	0	0	0	3	0	0	0	5	0	0	0
still-life-wastage	0	5	0	0	0	5	0	0	0	5	0	0	3	2	0	0	5	0	0	0
tpp	3	2	0	0	5	2	0	0	6	1	0	0	6	1	0	0	7	0	0	0
train	1	5	0	0	1	5	0	0	1	5	0	0	0	4	0	0	1	5	0	0
vrp	0	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0
$\sum$	32	31	0	13	46	34	0	0	47	36	0	0	50	32	0	0	57	28	0	0
2013																				
black-hole	0	0	1	0	0	0	1	0	0	0	1	0	2	0	1	0	4	0	1	0
cargo	3	0	0	0	0	0	0	5	2	1	0	0	0	2	0	0	0	5	0	0
celar	0	4	0	1	0	4	0	1	0	4	0	0	0	5	0	0	0	5	0	0
filters	4	1	0	0	4	1	0	0	4	1	0	0	5	0	0	0	4	1	0	0
fjsp	0	0	0	5	1	1	0	1	1	2	0	1	2	3	0	0	2	1	0	2
ghoulomb	0	0	0	3	0	1	0	1	0	1	0	1	3	0	0	0	3	0	0	0
javarouting	0	0	0	0	0	0	0	0	0	3	0	0	2	3	0	0	1	4	0	0
l2p	1	2	0	0	2	1	0	0	0	0	0	0	4	0	0	0	5	0	0	0
league	2	2	0	0	2	2	0	0	4	1	0	0	2	2	0	0	2	3	0	0
mario	2	2	0	0	3	1	0	0	5	0	0	0	5	0	0	0	4	1	0	0
nmseq	1	0	0	0	1	0	0	0	5	0	0	0	5	0	0	0	3	0	0	0
nonogram	2	0	0	0	5	0	0	0	5	0	0	0	2	0	0	0	3	0	0	0
on-call-rostering	4	1	0	0	3	2	0	0	3	1	0	0	2	3	0	0	2	3	0	0
pattern-set-mining	1	3	0	1	1	3	0	1	2	3	0	0	2	3	0	0	1	4	0	0
pentominoes-int	0	0	0	5	0	0	0	1	0	0	0	1	5	0	0	0	5	0	0	0
proteindesign12	0	4	0	1	0	4	0	1	0	4	0	1	2	3	0	0	3	2	0	0
radiation	5	0	0	0	5	0	0	0	4	1	0	0	3	0	0	0	4	0	0	0
rcpsp	2	3	0	0	1	4	0	0	1	4	0	0	0	5	0	0	4	1	0	0
rubik	3	0	0	0	3	0	0	0	3	0	0	0	1	0	0	0	5	0	0	0
vrp	1	4	0	0	1	4	0	0	1	4	0	0	1	4	0	0	1	4	0	0
$\sum$	31	26	1	16	32	28	1	11	40	30	1	4	48	33	1	0	56	34	1	2
2014																				
amaze	4	0	0	0	4	0	0	0	1	0	0	0	3	0	0	0	5	0	0	0
cyclic-rcpsp	4	0	0	0	0	0	0	5	4	0	0	0	0	5	0	0	3	2	0	0
elitserien	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0
fillomino	3	0	0	0	3	0	0	0	2	0	0	0	2	0	0	0	4	0	0	0
jp-encoding	5	0	0	0	5	0	0	0	5	0	0	0	0	5	0	0	0	5	0	0
liner-sf-repositioning	0	1	0	0	0	1	0	0	0	3	0	0	1	3	0	0	1	3	0	0
mario	1	2	0	0	2	2	0	0	5	0	0	0	4	1	0	0	3	2	0	0
mqueens	1	4	0	0	2	3	0	0	3	2	0	0	3	0	0	0	2	3	0	0
multi-knapsack	5	0	0	0	5	0	0	0	5	0	0	0	2	0	0	0	2	0	0	0
openshop	2	3	0	0	1	2	0	0	1	2	0	0	1	4	0	0	0	5	0	0
rectangle-packing	5	0	0	0	0	0	0	5	5	0	0	0	3	0	0	0	3	0	0	0
road-cons	0	5	0	0	0	5	0	0	0	5	0	0	1	4	0	0	0	5	0	0
ship-schedule	5	0	0	0	5	0	0	0	5	0	0	0	3	2	0	0	3	2	0	0
smelt	0	0	0	5	0	0	0	5	3	1	0	0	1	4	0	0	2	3	0	0
solbat	2	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4	0	1	0
spot5	2	3	0	0	2	3	0	0	2	3	0	0	0	5	0	0	0	5	0	0
stochastic-fjsp	4	1	0	0	4	1	0	0	4	1	0	0	2	3	0	0	4	1	0	0
stochastic-vrp	0	5	0	0	0	5	0	0	3	2	0	0	3	2	0	0	5	0	0	0
train	1	4	0	0	1	4	0	0	1	4	0	0	0	1	0	0	1	4	0	0
traveling-tppv	0	3	0	0	0	3	0	0	0	3	0	0	0	5	0	0	0	5	0	0
$\sum$	44	31	1	5	36	29	1	15	52	26	1	0	31	44	1	0	47	45	1	0
$\sum\sum$	107	88	2	34	114	91	2	26	139	92	2	4	129	109	2	0	160	107	2	2

Table 5. Effect of removing the redundant `cumulative`s, time limit 100s

	With redundant		No redundant	
	OPT	FEAS	OPT	FEAS
cargo	2	2	0	0
carpet-cutting	0	2	0	5
mspsp	5	1	4	2
rcpsp	1	4	1	4
rectangle-packing	4	0	5	0
smelt	3	1	3	1

Table 6. Effect of redundant `cumulative`s on the carpet-cutting problem and comparison with the Challenge results

	With redundant		No redundant		Ch2012_best
	obj	LB	obj	LB	obj
mzn_rnd_test.05.dzn	1563	1050	1155	1050	1191
mzn_rnd_test.10.dzn	noobj	3551	4894	3551	4520
mzn_rnd_test.14.dzn	noobj	1640	1982	1640	1937
mzn_rnd_test.16.dzn	1280	1075	1171	1075	1236
mzn_rnd_test.17.dzn	noobj	1862	2339	1862	2216

failed flattening instances from 26 to 4, and increased the number of instances with proven optimal solutions from 114 to 139, with a time limit of 5 minutes per instance of the MiniZinc Challenges 2012–2014.

To our surprise, with the new MILP translation the MILP solver IBM ILOG CPLEX 12.6.1 yields more optimal solutions than the 2014 Challenge winner, OR-Tools.

We are also exploring other improvements, including adding more facet-defining cuts, and exploiting new features of MILP solvers. We will also investigate the models' sensitivity to parameters such as the maximum size of a variable's domain for unary encoding. Another promising avenue is the application of pre-defined search strategies.

References

1. https://github.com/geoffchu/chuffed, 2015. Downloaded on 23 July 2015.
2. https://github.com/google/or-tools, 2015. Downloaded on 15 July 2015.
3. http://www.ampl.com, 2015.
4. 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.
5. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.
6. G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, 2015.

7. 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.
8. IBM Software. IBM ILOG CPLEX optimizer. Data sheet, IBM Corporation, 2014.
9. K. Marriott and P. J. Stuckey. The MiniZinc tutorial, 2014.
10. 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.
11. 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.
12. A. Schutt, Th. Feydy, P. J. Stuckey, and M. G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011.
13. 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.
14. P.J. Stuckey, R. Becket, and J. Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.
15. 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.