UPPSALA
UNIVERSITET

IT 21 015
Examensarbete 45 hp
Mars 2021

The MiniZinc-SAT Compiler

Hendrik Bierlee

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The MiniZinc-SAT Compiler

Hendrik Bierlee

Combinatorial (optimization) problems occur in nature and society. Constraint modeling languages such as MiniZinc allow the user to declaratively specify such problems in terms of its decision variables and constraints. Subsequently, the MiniZinc model can be compiled into an equivalent (Boolean) SAT formula to be solved by a SAT solver. This thesis reports on the design, implementation and experimental evaluation of the new MiniZinc-SAT compiler, which supports single or even mixed usage of three distinct integer variable encoding methods.

Handledare: Peter J. Stuckey, Jip J. Dekker (co-supervisor)
Ämnesgranskare: Justin Pearson
Examinator: Mats Daniels
IT 21 015
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to thank my supervisor Prof. Peter J. Stuckey for all the help and guidance I have received during the project, showing me at each weekly meeting that things were not as simple as I had thought, but rather simpler.

I would like to thank as well my co-supervisor, PhD candidate Jip Dekker, for his daily help and efforts towards this project. His experience, patient explanations and eye for detail have been a driving factor.

I would like to thank my reviewer at Uppsala University, Prof. Justin Pearson, for his remote support throughout this singular period. All my questions and issues were resolved so efficiently through our email conversations that it was as if I was conducting my thesis from the next room over, instead of from the next hemisphere.

Finally, I would be amiss if I did not mention and thank Prof. Pierre Flener at Uppsala University, without whose truly inspirational and expertly crafted courses on Constraint Programming and Combinatorial Optimization I would still be wandering around aimlessly through the fields of Computer Science.

Contents

1 Introduction **1**

2 Background **2**
- 2.1 Constraint Programming . . . 2
- 2.2 SAT problems and SAT solvers . . . 2
- 2.3 Booleanizing an example Sudoku model . . . 2
  - 2.3.1 Sudoku as a CSP . . . 3
  - 2.3.2 Sudoku as a SAT model . . . 5
- 2.4 Other relevant problem classes . . . 6
- 2.5 Thesis structure . . . 7

3 Related Work **8**

4 Booleanization **10**
- 4.1 Integer encoding techniques . . . 10
  - 4.1.1 Integer encoding fundamentals . . . 10
  - 4.1.2 Sharing variables with Common Subexpression Elimination (CSE) . . . 13
  - 4.1.3 Dynamic encodings . . . 14
  - 4.1.4 Channeling constraints . . . 16
- 4.2 The `sat` redefinition library . . . 19
  - 4.2.1 Dynamic encodings in MiniZinc . . . 20
  - 4.2.2 FlatZinc built-ins: Boolean . . . 24
  - 4.2.3 FlatZinc built-ins: integers . . . 25
  - 4.2.4 FlatZinc built-ins: sets . . . 30
  - 4.2.5 Global constraints . . . 30
  - 4.2.6 Optimization . . . 31
  - 4.2.7 Testing . . . 32
- 4.3 The SAT solver interface . . . 33
  - 4.3.1 From CNFlatZinc to (W)DIMACS . . . 33
  - 4.3.2 SAT solution parser . . . 34
  - 4.3.3 SAT solution output . . . 35

5 Evaluation (experiments and discussion) **37**
- 5.1 Evaluation of two `mzn-sat` features . . . 37
  - 5.1.1 Channeling constraints . . . 37
  - 5.1.2 Set and reified set constraints . . . 39
- 5.2 Evaluation of `mzn-sat` and existing back-ends . . . 41
  - 5.2.1 Per-solver evaluation . . . 41
  - 5.2.2 Per-model evaluation . . . 44
  - 5.2.3 Per-instance evaluation . . . 44

6 Conclusion **49**

7 Future Work **50**

A A less redundant binary encoding scheme for signed integers 51

B One-way channeling constraints 51

C The `bimander` encoding for the at-most-one cardinality constraints 52

D The `mimander` encoding for multiple overlapping at-most-one cardinality constraints 53

E Sorting networks for general Boolean cardinality constraints 54

F Binary Decision Diagrams (BDDs) 58

G FlatZinc built-ins: arithmetical (plus, times, div) integer constraints 62
G.1 Dual encodings 62

H Constraining Boolean cardinality constraints for a non-fixed count c 64

I The `--dimacs-idn-mode` flag 64

J Evaluation of different `mzn-sat` default encodings and backend Max-SAT solvers 65

K Per model results 67

1 Introduction

In combinatorial (optimization) problems, the goal is to find an (optimal) object from a finite but potentially huge set of objects. Such problems occur in nature and society, and have many applications.

Constraint Programming (CP) is a paradigm for solving combinatorial problems with a declarative approach. Instead of specifying the solving algorithm step-by-step (the imperative approach), the programmer models the problem in terms of its variables and constraints. It is up to a solver to find a variable assignment that satisfies the constraints, and optimizes the objective function if one is given. CP has been applied in domains such as scheduling, planning, vehicle routing, and so on [1].

There are many solver technologies (e.g., backtracking, constraint propagation, local search, etc.) and many solvers that implement each technique. The choice of solver or solving technology can have drastic consequences for solving speed. Instead of translating models from language to language, MiniZinc was designed to be a solver-independent modeling language by interfacing with different solvers [2].

One of the most powerful classes of solvers are the Boolean satisfiability problem ((Max-)SAT) solvers, but MiniZinc does not (natively) support this type of solver yet. Since SAT problems are to be expressed only in Boolean logic, an interface is required that first compiles the high-level CP model down to an equivalent low-level SAT problem. While this process adds complexity and compilation time, on many instances the SAT solver still might prove faster, even with this overhead, than other solvers technologies.

Currently, MiniZinc can interface with some (Max-)SAT solvers by first compiling to other CP languages which do have a CP-to-SAT compiler (e.g., Picat-SAT). However, an actual MiniZinc-SAT compiler would give MiniZinc more control over the translation, access to any SAT solver (after adding the FlatZinc interface), and might be more efficient because it would avoid the overhead of intermediary translation (since the resulting Picat model might be inferior to the original MiniZinc model). Furthermore, this project will allow us to assess the current state of CP-to-SAT compilation, and implement a fresh approach.

2 Background

2.1 Constraint Programming

In Constraint Programming, a constraint satisfaction problem (CSP) consists of a set of variables $\mathcal{X}$, with each $x \in \mathcal{X}$ over some domain $D(x)$. A set of constraints $\mathcal{C}$ expresses relationships between the variables. An assignment $\mathcal{A}(\mathcal{X})$ specifies a value for each variable and is a solution to the CSP if the values adhere to their variable's domains and constraints. For convenience, $D_l(x)$ and $D_u(x)$ refer to the lower and upper bound of x, respectively.

A constraint optimization problem (COP) is an extension of a CSP where an objective function attributes an objective value to an assignment. The solver will aim to find an assignment that minimizes or maximizes the objective value.

In CP, variables can have general domains (e.g., integer, float, set, etc.), and the constraints can be existential (x is one of a set of values), arithmetic ($x > y + 5, xy < z$) or even *global*. An example of a global constraint is `all_different` on a set X of integer variables, which constrains all the values in X to be be pairwise different.

2.2 SAT problems and SAT solvers

The SAT problem, which is a subclass of a CSP, is the first ever problem to be proven NP-complete [3]. It consists of a set of *propositional* variables $\mathcal{X}$, and a set of constraints $\mathcal{C}$ that expresses a Boolean formula. A solution of the problem is an assignment $\mathcal{A}(\mathcal{X})$ that evaluates the formula to true. If this assignment exists, the problem is satisfiable, if it does not, unsatisfiable. Since many real world problems can be encoded as SAT problems, many SAT solvers such as `Lingeling` [4] have been implemented to solve them efficiently, using methods like the DPLL algorithm [5]. For most SAT solvers, the input needs to be normalized to Conjunctive Normal Form (CNF). In CNF, every constraint $c \in \mathcal{C}$ is a Boolean clause, which are disjunctions of Boolean literals, each of which is a variable x or its negation $\overline{x}$.

A Max-SAT problem is a subclass of COP, and an extension of SAT, aims to maximize the number of satisfied clauses. Clauses can be given weights w to prioritize them, or create hard clauses with infinite weight that must be satisfied. Such problems can be tackled through repeated calls to SAT solvers, and this forms the basis for many Max-SAT solvers such as `MaxHS` [6], or the popular modular Max-SAT solver `Open-WBO` [7] on which many other experimental solvers are based. Complete Max-SAT solvers limit themselves to only optimal solutions, while incomplete Max-SAT solvers such as `Open-WBO-inc` [8] try to find the best assignment in a limited amount of time, even if it is sub-optimal.

2.3 Booleanizing an example Sudoku model

How to actually translate a CP model to SAT, and why it might be beneficial to do so, can be made clear through the use of an example. Let's create a basic

encoding of the well-known Sudoku problem, which has been studied before in the context of SAT solvers [9]. The goal of this newspaper puzzle is to fill in the blank squares in a 9×9 grid with digits from 1 through 9, such that no digit appears twice in any column, row or 3×3 subgrid, which form the *regions* of the grid. In other words, in a correct solution every region contains every number from 1 through 9. Some hints are given at the start, which constitute the problem instance.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1: A typical Sudoku instance (taken from `https://en.wikipedia.org/wiki/Sudoku`)

2.3.1 Sudoku as a CSP

Modeling this as a CP problem, we recognize that the 81 squares are our decision variables. Each variable has an integer domain of 1 to 9. The filled-in squares (such as the 5 in the top left corner) become fixed variables (having a domain of size 1). A model is written in a CP specification/modeling language, such as MiniZinc. As discussed in section 1, MiniZinc is solver-independent modeling language, and a possible MiniZinc model of the Sudoku CSP is as follows:

Listing 1: A MiniZinc model for a 9×9 Sudoku

```
% X[i,j] denotes the square at row i, column j
array[1..9,1..9] of var 1..9: X;
constraint forall(i in 1..9)(
  all_different(j in 1..9)(X[i,j])
);  % Rows
constraint forall(i in 1..9)(
  all_different(j in 1..9)(X[j,i])
);  % Columns
constraint forall(i,j in 1..3)(
  all_different(p, q in 1..3)(X[3*(i-1)+p,3*(j-1)+q])
);  % 3x3 subgrids
```

First, the decision variables are denoted with the keyword `var`, followed by a set representing the domain. We can instantiate the required number of 81 integer variables by constructing a 2D-array `X` with dimensions 9×9 of variables, each having a domain of $[1, 9]$. Next, constraints can be posted using the `constraint` keyword. In this case, the `forall(...)(...)` constraint posts a conjunction of constraints, which allows us to loop over one or more iterators (within the first parentheses) to create 27 `all_different` constraints (within the second parentheses) over the rows, columns and sub-grids. The effect is that all variables within the same region must be pairwise different. The `all_different` has the same iterator syntax.

At this point, we have declared a general model for any Sudoku instance. Normally, the dimension of the square grid (9) would be specified as a parameter `n`, so that we can solve instances of any size (as long as $\sqrt{n} \in \mathbb{N}$). Note that we did not specify anywhere *how* the model should be solved. This is the domain of the solver. Before we hand the model over to the solver, we need to fix some of the elements of X according to the hints of our instance:

```
X=[|
  5, 3, _, _, 7, _, _, _, _|
  6, _, _, 1, 9, 5, _, _, _|
  _, 9, 8, _, _, _, _, 6, _|
  8, _, _, _, 6, _, _, _, 3|
  4, _, _, 8, _, 3, _, _, 1|
  7, _, _, _, 2, _, _, _, 6|
  _, 6, _, _, _, _, 2, 8, _|
  _, _, _, 4, 1, 9, _, _, 5|
  _, _, _, _, 8, _, _, 7, 9|
|];
```

At this point, we have a model that MiniZinc can feed to any of its interfacing solvers. It is able to do so through an intermediate process known as *flattening*, which breaks down and transforms the model from solver-agnostic MiniZinc into an equivalent model in solver-specific FlatZinc, which is a subset of the MiniZinc language. For instance, flattening will unroll our loops into 27 `all_different` constraints, or introduce new variables to model complex constraints in simpler terms.

How much works this is depends on the solver. If a solver supports the `all_different` constraint natively (such as Gecode), then these constraints can be left as-is. Otherwise, these constraints have to be replaced by their low-level definition. More complex is the case of linear solvers (MIP), which require that all constraints are linear, which means that during flattening the non-linear constraints must be linearized.

Additionally, if the solver does not accept FlatZinc as input, another interfacing layer is needed to interface to the solver – and from solver, when we parse its solutions. Despite the engineering and performance overhead, this system does provide the modeler with the freedom to experiment with different solving techniques.

2.3.2 Sudoku as a SAT model

The flattening procedure for SAT-solvers will be quite drastic since such solvers do not understand `all_different` constraints, or even integers, or even all of Boolean algebra. All we have to work with are Boolean variables, conjunctions ($\wedge$), disjunctions ($\vee$) and negations (either $\neg b$ or $\bar{b}$). To represent all integer variables $x_{i,j}$ at each row i and column j as Boolean variables, we need to create an encoding. One way to do this is by introducing one Boolean variable per value in the domain of $x_{i,j}$. This is known as the *direct encoding* of the integer variables. In our case, this means we need 9 fresh Boolean variables $b_{i,j,1}, b_{i,j,2}, \ldots, b_{i,j,9}$ to represent one blank square. If $b_{i,j,5}$ is assigned true ($b_{i,j,5} = 1$) in the solution, we interpret the square as having the value of 5 (or in general, $b_{i,j,k} \Leftrightarrow x_{i,j} = k$). This means that for a worst-case (blank) Sudoku instance, we get $9 \times 9 \times 9 = 729$ Boolean variables.

Since integer variables cannot have multiple values simultaneously (or be without value), we have to add a cardinality constraint to enforce that exactly one of the 9 encoding variables must be true. This is the same as saying that at least one *and* at most one variable must be true. The former is quite easy, by posting for each row i and column j the single clause of $b_{i,j,1} \vee b_{i,j,2} \vee \cdots \vee b_{i,j,9}$. The latter constraint turns out to be a well-studied problem, and many smart and efficient encodings have been proposed and compared. A simple way of doing it would be to add a clause for each possible pair of variables which says that one or the other must be false ($(\overline{b_{i,j,1}} \vee \overline{b_{i,j,2}}) \wedge (\overline{b_{i,j,1}} \vee \overline{b_{i,j,3}}) \wedge \cdots$), which leads to 32 clauses of size 2 for a single integer with a domain of size 9.

With the direct encoding scheme, it turns out that an `all_different` constraint is quite natural to enforce. This constraint encoding works well for feasible instances like ours, although other variants exist that scale better [10]. Let's look at the constraint for the first column: `all_different(X[..,1])`. The direct encoding of this single array of integer variables form a new "encoding" grid of Boolean variables. The first row of the grid represents the integer variable of which we already know the value (5), so it will contain zeroes except in the 5th position. The same goes for the second variable in the first Sudoku's column (6), which is the second row in our encoding grid. Four of the integers are not fixed, so their rows contain fresh non-fixed Boolean variables `b`:

```
          123456789
      5 -> 000010000
      6 -> 000001000
X[3,1] -> bbbbbbbbb  -> FFaFFFbcd
      8 -> 000000010         =
      4 -> 000100000  AtMost1([a,b,c,d])
      7 -> 000000100         =
X[7,1] -> bbbbbbbbb  7/12 vars/clauses
X[8,1] -> bbbbbbbbb
X[9,1] -> bbbbbbbbb

AtMost1([0,0])
```

Now, posting an `all_different` constraint is essentially the same as saying that there can be at most one variable that takes the value of 1, one variable for the value of 2, and so on. Since the first column of the encoding grid tells us exactly which of the integer variables in the first Sudoku column have a value of one (e.g., $a \Leftrightarrow X_{3,1} = 1$), we can use our at-most-one cardinality constraint again to constrain the first column in the encoding grid to have at most one true literal:

$$\begin{aligned}
&\texttt{at_most1}([0, 0, b_{3,1,1}, 0, 0, 0, b_{7,1,1}, b_{8,1,1}, b_{9,1,1}]) \\
&= \texttt{at_most1}([b_{3,1,1}, b_{7,1,1}, b_{8,1,1}, b_{9,1,1}]) \\
&= (\overline{b_{3,1,1}} \vee \overline{b_{7,1,1}}) \wedge (\overline{b_{3,1,1}} \vee \overline{b_{8,1,1}}) \wedge (\overline{b_{3,1,1}} \vee \overline{b_{9,1,1}}) \\
&\wedge (\overline{b_{7,1,1}} \vee \overline{b_{8,1,1}}) \wedge (\overline{b_{7,1,1}} \vee \overline{b_{9,1,1}}) \wedge (\overline{b_{8,1,1}} \vee \overline{b_{9,1,1}})
\end{aligned}$$

This particular column has 4 non-fixed variables, for which the pairwise encoding produces 6 clauses. If we do this for each column of the encoding grid, then the integers in the first column of the Sudoku will effectively be constrained to be all different. All in all, we use one at-least-one and one at-most-one constraint per non-fixed integer, and (at worst) 9 at-most-one constraints per region. For our instance, which contains 51 non-fixed integer variables, flattening (with all standard flattening optimizations disabled) will produce a SAT model of 459 Boolean variables and 3292 clauses, which a state-of-the-art SAT solver like Lingeling solves in 4 milliseconds. With built-in MiniZinc optimizations, in which for instance the domains of the integers are reduced before we perform our encoding method, the model contains only 153 Boolean variables and 701 clauses, which takes Lingeling 2 milliseconds. In comparison, the CP solver Gecode takes 65 milliseconds to solve the model.

Of course, in addition to being too small a model to draw any meaningful conclusions from, it is also an unfair comparison because the compilation time (to generate the SAT model) is not taken into account, which is the other side of the coin for this optimization strategy: we want to create an efficient model, efficiently. However, this does show the general method and potential of a CP-SAT compiler.

2.4 Other relevant problem classes

Mixed Integer Programming (MIP) problems are a subclass of CP problems, where the constraints are all linear:

$$\sum_{i=1}^{n} a_i x_i \ \# \ c \tag{1}$$

Which constrains n terms (coefficients $a_1, a_2, \ldots, a_n$ and variables $x_1, x_2, \ldots, x_n$) and right-hand side constant c, with $\#$ being any comparison comparator ($=, \leq, \neq, \ldots$). There already exists a linearization library for MiniZinc for a MIP

solver interface [11][1]. In Integer Programming (IP) problems, the variables are integers, in Pseudo-Boolean (PB) Programming problems, the variables are binary integers, having a domain of $\{0, 1\}$.

We can calculate the lower bound of the left-hand side of linear constraints (and linear expressions in general) with:

$$D_l\left(\sum_{i=0}^{n} a_i x_i\right) = \sum_{i=0}^{n} \begin{cases} a_i D_l(x_i) & \text{if } a_i > 0 \\ a_i D_u(x_i) & \text{otherwise} \end{cases} \tag{2}$$

The upper bound is symmetrically calculated. For PB constraints, D_l and D_r of the left-hand side are the sum of the left-hand side negative or positive coefficients, respectively.

2.5 Thesis structure

The structure of the rest of this thesis is as follows: In section 3, we will discuss related projects such as other general CP-SAT compilers. In section 4, we will explain in detail what methods are used for `mzn-sat`, the new MiniZinc-SAT compiler. In section 5, we will show and discuss the experimental results with which the system was evaluated. In 6, we conclude our finding and in 7 we advice on future work directions for the project. Noteworthy but unrealized ideas are listed in appendix 7 or in footnotes.

[1] The linearization library was used as a template for the `sat` library, but very little code from this project is left over in the final implementation.

3 Related Work

This section covers existing universal CP-SAT compilers and their integer encoding scheme, which is their most distinguishing feature. Other related work for this project is on encodings for specific constraints, which will be introduced in section 4.

`FznTini` [12] (binary encoding, two's complement) Laying the claim to be the first universal CP-SAT compiler, `FznTini` takes MiniZinc as base CSP language. Apart from competitive results on some models compared to CP solvers G12/FD [13] and Gecode, it also shows that Boolean CSP models still retain their performance under compilation. A better encoding for the `Cumulative` global constraint was later added in `FznTini+`. While unmaintained and only compatible with older versions of `FlatZinc`, the available binary still solves many small problems.

`Sugar` [14] ((named) order encoding) While it lacks support for non-linear constraints, it ranked middling overall in CSP 2008 and 2009 competitions, and placed in top positions for some global constraint categories. The choice for order encoding has been supported in their own work comparing it to other encodings [15], and also has theoretical basis established in independent work [16]. Since then development has continued, notably improving the PB to SAT encoding [17].

`Azucar` [18] (compact order encoding) Compact order encoding is a generalization of named order encoding and binary encoding, which uses a numeral system with an arbitrary base. The result is much compacter than named order encoding, where each value of each variable has its own encoding variable, but it retains the propagation strength of order encoding. It is much faster than `Sugar` in proving instances to be unsatisfiable, and supports non-linear constraints.

`Picat-SAT` [19] (binary encoding, sign-and-magnitude) `Picat` [20] is a CP modeling language which now provides a `sat` module. Despite the negative research results on binary encoding, and it not maintaining arc-consistency, the results see `Picat-SAT` significantly outperforming `Sugar/Azucar` on some models. The authors conclude that binary encoding is still a viable option due to its compactness and roots in computer hardware design. `Picat` is compatible with `FlatZinc`, which allows us to compare our own results with `Picat-SAT`.

Lazy Clause Generation (LCG) [21] Lazy clause generation is a technique where traditional CP is supported by a SAT solver. For example, a full encoding of constraints involving a variable x with a large domain yield a large amount of clauses, so instead first finite domain propagation will explore search branches that limit the domain (say $x \leq 5$). Only then, clauses of the form $x \leq 5 \Rightarrow \cdots$

are generated and the SAT solver is run. If the search branch is abandoned (backtracking) because it does not lead to a satisfiable assignment, then $x \not\leq 5$ can be added so that the added clauses no longer propagate in the SAT solver. However, this does not mean any clauses have to be retracted, since they are still globally true and might aid the next SAT solver process in other ways. So, the solver does not need to be restarted and retains all it has learned about the problem so far. `Chuffed` is a LCG solver which builds a full order encoding and then generates an additional direct encoding on-demand. It interfaces with MiniZinc, so it will be a good candidate to compare against, as both use SAT solver techniques in different ways.

Sp-Or encoding [22] This experimental encoding for a limited CSP subset employs both direct and order encoding, redundantly. Since the first is suited to equalities and the second to inequalities, the Sp-Or encoding improves on models that have both types of constraints.

Satune [23] In this work, it is observed that the best choice of encoding for a given problem can be inferred by running a search progress (in their case, using simulated annealing) on smaller problems in the same domain. A specialized encoder is synthesized and fine-tuned to great effect. Currently, `Satune` is still limited to a domain specific language in which the problem needs to be expressed.

4 Booleanization

The goal of the Booleanization process is to convert FlatZinc constraints and variables into an equivalent SAT-model in CNF. We discuss the Booleanization process in three parts: First, we introduce fundamental integer encoding techniques in section 4.1. Then, we describe the `sat` redefinition library where these techniques are applied to create a flattened model in section 4.2. Finally, we discuss the interfacing layer between MiniZinc and SAT solver in section 4.3.

4.1 Integer encoding techniques

4.1.1 Integer encoding fundamentals

Since a SAT problem is restricted to Boolean variables, the model's integer variables are mapped onto their own set of Boolean variables. Given an integer x with (possibly non-contiguous) domain $D(x) = \{d_1, d_2, \ldots, d_m\}$, which is ordered ($d_i < d_{i+1}$), we implemented three encoding methods. In Table 1, we show an example assignment of each encoding on an integer variable with a small domain. To formally treat and define them we introduce encoding functions which, after x has been encoded, are used to access the encoding variables.

Direct encoding [21, 24, 25] The direct encoding function f for x direct encoded with m Boolean variables $B = \{b_{d_1}, b_{d_2}, \ldots, b_{d_m}\}$ is:

$$f(x = v) = \begin{cases} b_v & \text{if } v \in D(x) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The encoding variables are interpreted such that $f(x = v) \Leftrightarrow x = v$. The intuitive advantage of modeling with this encoding is that any variable will inform us of the *equality* of x to some value. This makes it easy for solvers to eliminate single values from domains, increasing propagation strength [26]. It is also known as value, onehot-, or sparse encoding. The `exactly_1` constraint (see section C) is posted on B so that the solver will be forced to assign x a single value.

In MiniZinc, an array has an index set, which is a closed interval $[l, u]$ with arbitrary bounds (i.e., the array is not necessarily 0- or 1-indexed like in other programming languages). By setting $l = D_l(x)$ and $u = D_u(x)$, we can access $f(x = v)$ directly by using v as the index. The array contains extra information (on the bounds of the integer its encoding) that is not available when looking at any literal by itself. The SAT solver does not need this information to solve the problem correctly. Unlike domains, index sets have to be contiguous, so any gaps G in the domain ($G = \{i : \forall(D_l(x) < i < D_u(x)), i \notin D(x)\}$) are represented by fixed `false` literals in the array. Out-of-bound accesses of the array are allowed and return `false` as well, which is consistent with the definition.

When the direct encoding is created, we can avoid checking $v \in D(x)$ for every value in $D_l(x) \leq v \leq D_u(x)$ by converting the set $D(x)$ to an array of

ranges (intervals), e.g., $\{1, 2, 3, 8, 9, 10\}$ becomes $([1, 3], [8, 10])$. This allows us to instantiate an array of 3 variables, followed by 4 false literals, followed by 3 more variables safely and without checks or redundant variables. MiniZinc stores sets internally in that format anyway and has a built-in function (`set_to_ranges`) to make these available. Going through the intervals sequentially saves us the $|D(x)|$ domain checks [2].

Order encoding[14, 17, 25] The order encoding function f for x order encoded with $m-1$ Boolean variables $b_{d_2}, b_{d_3}, \ldots, b_{d_m}$ is:

$$f(x \geq v) = \begin{cases} b_v & \text{if } v \in D(x) \\ 1 & \text{if } v \leq D_l(x) \\ 0 & \text{if } v > D_u(x) \\ f(x \geq v+1) & \text{otherwise} \end{cases} \tag{4}$$

The encoding variables are interpreted such that $x = d_1 + \sum_{i=2}^{m} f(x \geq v)$. This encoding will tell us the *inequality* of x to some value with $f(x \geq v) \Leftrightarrow x \geq v$ or $\neg f(x \geq v) \Leftrightarrow x < v$. To determine the equality of x to v, we need both $f(x \geq v) \wedge \neg f(x \geq v+1)$. Furthermore, while this is not beneficial in all scenarios [25], we do constrain the encoding to always be ordered ($f(x \geq v) \Rightarrow f(x \geq v-1)$). The first element of the MiniZinc array holding the variables is always a fixed padding element (1) to line up the index set with the index set of the direct encoding. Order encoding is also known as unary, ladder, regular or thermometer encoding.

Binary encoding [19, 12, 25] For the binary encoding function f for x binary encoded with n Boolean variables $b_0, b_1, \ldots, b_{n-1}$ is:

$$\begin{cases} f_n(x, i) = \text{undefined} & \text{if } i < 0 \\ f_n(x, i) = b_i & \text{if } 0 <= i <= n-2 \\ f_n(x, i) = b_{n-1} & \text{otherwise} \end{cases} \tag{5}$$

In traditional unsigned binary encodings, every bit has a significance or *value* of 2^i based on its position i, which means the total value of x is interpreted from the assignment of the encoding as the sum of the values of the true bits:

$$x = \sum_{i=0}^{n-1} 2^i f(x, i) \tag{6}$$

This means the encoding range will be in the interval $[0, 2^n - 1]$, where we will call the lower and upper limits the *floor* and *ceiling* of the encoding, respectively. The obvious problem is that since the floor is always 0 (where each

[2] This could be implemented for order encoding as well, but I couldn't quite get this to work due to time constraints.

bit is 0), we cannot represent negative domains, which means no signed integers. So, instead, we will use the well-known two's complement scheme [27]:

$$x = \left(\sum_{i=0}^{n-2} f(x,i)2^i\right) - f(x,n-1)2^{n-1} \tag{7}$$

This scheme works the exact same as the unsigned binary encoding scheme, except that the last bit (with position $i = n-1$, also known as the *sign* bit) is subtracted instead of added. This means that n bits have an encoding range of $\left[-2^{n-1}, 2^0 + 2^1 + \cdots + 2^n - 2 = 2^{n-1} - 1\right]$. While there are other binary representations such as sign-and-magnitude (as used in `picat-sat`) or one's complement, the two's complement retains more mathematical properties as we will see[3].

The minimal number of required n bits for x thus generally depends on which bound of x has the greater absolute value, since that will decide whether the floor or the ceiling is the bottleneck:

$$n = \begin{cases} \lceil \log\left(|D_u(x) + 1|\right) + 1 \rceil & \text{if } |D_l(x)| < |D_u(x)| \vee \left(D_l(x) = D_u(x) \wedge D_u(x) \geq 0\right) \\ \lceil \log\left(|D_l(x)|\right) + 1 \rceil & \text{otherwise} \end{cases} \tag{8}$$

Whenever we write the binary encoding function without n, we imply that n is this minimum number.

There will likely be redundancy in the encoding, i.e. assignments that represent a value not in $D(x)$. Unless both bounds line up exactly with the encoding range, we still have to create inequality constraints $x \leq D_u(x)$ and $x \geq D_l(x)$. In addition, for non-contiguous domains, the gaps should be eliminated with $\bigwedge_{g \in G} x \neq g$. See section 4.2.3 for how these constraints are finally handled.

This compact encoding saves on variables and clauses, which becomes especially important as domains grow in size. The binary encoding has a disadvantage when it comes encoding small domains with large numbers is somewhat inefficient, since the encoding is centered around the value of 0. For example, $D(x) = [1000, 1002]$ requires 11 variables compared to 3 for the direct encoding. Once the domains grow, the binary encoding will quickly catch up again in terms of efficiency.

However, since flipping a single Boolean removes half of the values from the domain, it is generally harder for solvers to propagate on this encoding. Also, most constraints will be harder to express for this encoding (e.g., the constraint $x \neq 2$ requires for the direct encoding a single unit clause $\neg f(x = 2)$, while for a binary encoding of size $n = 4$ it requires a clause of size n that eliminates the specific permutation of variables that represents 2 with $\neg f(x,0) \vee f(x,1) \vee \neg f(x,2)$). Fortunately, the two's complement scheme provides us some interesting properties to make this manageable:

[3]Before two's complement was adopted, a different signed binary encoding was in place that I have not encountered elsewhere. It is impractical to use for most constraints, but does have the least amount of redundancy (see appendix A).

- Addition and subtraction of binary encodings work as expected (e.g., $-4+2 = 0001 + 0100 = 0101 = -2$). Note that the binary bits from left to right are to be read from least to most significant.
- Flipping the sign of x can be done by simply flipping every bit and adding one, which is also known as taking the two's complement of the encoding.
- The two's complement binary encoding function f can be converted to what we will call the *orderable* binary encoding function f' by flipping the sign bit:

$$\begin{cases} f'(x,i) = \neg f(x,i) & \text{if } i = n-1 \\ f'(x,i) = f(x,i) & \text{otherwise} \end{cases} \tag{9}$$

  In some cases, we convert the two's complement encoding to the orderable binary encoding so that the sign bit (which is normally deducted from the total) has the same semantics as the other bits (a value of 2^i). The orderable binary encoding is also in reverse lexicographical order, which means that $x \geq y$ holds if and only if:

$$\begin{aligned} (f'(x,n-1), f'(x,n-2), \ldots, f'(x,1), f'(x,0)) \geq_{\text{lex}} \\ (f'(y,n-1), f'(y,n-2), \ldots, f'(y,1), f'(y,0)) \end{aligned} \tag{10}$$

  E.g., $2 \geq -4 \Leftrightarrow \texttt{reverse}(010\overline{0}) \geq_{\text{lex}} \texttt{reverse}(000\overline{1}) \Leftrightarrow 1010 \geq_{\text{lex}} 0000$. This is useful for creating inequality constraints.
- A binary encoding of size n can be extended in size without changing its value by repeating the sign bit:

$$\begin{aligned} (f(x,0), f(x,1), \ldots, f(x,n-2), f(x,n-1)) = \\ (f(x,0), f(x,1), \ldots, f(x,n-2), f(x,n-1), f(x,n-1)) \end{aligned} \tag{11}$$

From now on, we can use the encoding functions $f(x = v)$, $f(x \geq v)$ and $f(x,i)$ to unambiguously refer to the associated Boolean variable of the direct, order or binary encoding of x, as well as to their negations with $f(x \neq v) = \neg f(x = v)$ and $f(x < v) = \neg f(x \geq v)$.

4.1.2 Sharing variables with Common Subexpression Elimination (CSE)

One crucial feature for encodings is to avoid creating the same encoding multiple times for the same integer variable. Not only is this redundant, but also incorrect if the constraints are split over the variables and there is no channeling between them. Instead, if there are two or more constraints on the same (values of) an

Table 1: All assignments of an integer variable x with domain $D(x) = \{-2, -1, 2, 3\}$ with the (only valid) corresponding assignments of the MiniZinc implementation of the direct, order and binary encoding represented as a bit string. The index sets of the arrays are $[2, 7]$, $[2, 7]$ and $[1, 3]$, respectively. When the encoding was constructed, some of the elements are fixed ($f(x = 5) = f(x = 6) = 0$ and $f(x \geq 2) = 1$) and some share the same variable ($f(x \geq 5) = f(x \geq 6) = f(x \geq 7)$). Even though at first glance it seems that a domain $D(x)$ of size 4 could be represented with 2 bits, the upper bound $D_u(x)$ of 3 pushes the minimal amount of required variables for B_{binary} in two's complement encoding to 3 (see equation 8).

$\mathcal{A}(x)$	$\mathcal{A}(B_{\text{direct}})$	$\mathcal{A}(B_{\text{order}})$	$\mathcal{A}(B_{\text{binary}})$
-2	`100000`	`100000`	`011`
-1	`010000`	`110000`	`101`
2	`000010`	`111110`	`010`
3	`000001`	`111111`	`110`

integer variable, they should act on the same variables and not instantiate new ones.

The encodings are created using an encoding construction function (e.g., with `order_encode(var int: x)`) from within the constraint (it is often the first step). If multiple constraints are posted on `x`, then `order_encode` could be called multiple times on the same input variable. Fortunately, Common Subexpression Elimination (CSE) [28] replaces repeat calls to the same function with a reference to the result from the first call. This techniques is popular in programming languages where functions do not have side-effects.

4.1.3 Dynamic encodings

One novel part of the project is the ability for the user or the library to choose the encoding, or mix different encodings, for the CP problem, which are techniques which have been shown to be valid in isolation [25, 22]. Here, we generalize this idea to a framework suitable for the full CP domain, and the three most popular encodings as defined in section 4.1.1.

To illustrate: in the Sudoku model in section 2.3, the direct encoding of an array of integer variables X lends itself well to the `all_different` constraint. However, other constraints (such as `array_int_max`, see section 4.2.3) are better off using the order encoding. If X is subject to both constraints, our choices are to either have two different encodings of X, or to add different versions of constraints for all encodings. What is best depends on whether the overhead of a re-encoding of X performs better or worse than an order encoded version of `all_different`. This is why our system accommodates both approaches.

The re-use approach We will call a constraint that supports multiple encodings a dynamic constraint. One major class of constraints that can be made

fully dynamic (support all three of or encodings) is the linear constraint. For any linear expression $c + \sum_{i=0}^{n} a_i x_i$ (where x_i are bounded integer variables and a_i and c are fixed coefficients), we can expand the i'th integer variable into multiple pseudo-Boolean terms (one per encoding variable) using any of our desired encoding as follows:

$$a_i x_i = \begin{cases} \sum_{j \in D(x_i)} a_i j f(x_i = j) & \text{if direct} \\ a_i d_1 + \sum_{j \in D(x_i) \setminus \{D_l(x)\}} a_i f(x_i \geq j) & \text{if order} \\ \left(\sum_{j=0}^{n-2} a_i 2^j f(x, j)\right) - a_i 2^{n-1} f(x, n-1) & \text{if binary (where } n \text{ is the binary encoding size)} \end{cases} \tag{12}$$

This approach is in part based on the MiniZinc QUBO integration library [29]. This is integrated for the linear objective (see section 4.2.6) and for linear constraints (see section 4.2.3), producing PB constraints that can be booleanized using BDDs (see appendix F for our own legacy implementation) or by any other techniques available in PBLib (an external library for encoding PB constraints to SAT; see section 4.3.1).

Since the encoding can be applied independently per term, a mix of encodings is also possible. This is ideal, since one variable might be encoded differently from another, while others might not have any encoding yet. In the last case, the best choice depends on if SAT solvers prefers BDDs generated from PBs from one encoding over the other. Direct encoding produces many terms and big coefficients (at most $d_m a_i$), while for order encoding the coefficients will be constant and small (a_i). For binary encoding, the number of terms will be much smaller, which is a very attractive feature for the flattening time. Its coefficients are of medium size (at most $2^{\lceil \log d_m - d_1 + 1 \rceil} a_i = \frac{1}{2}(d_m - d_1) a_i$) [4].

The re-encoding approach An advantage of the re-encoding system is that it can be applied generally, while the re-use of an encoding requires a new version of the constraint for that encoding. Furthermore, in some cases a re-encoding might be the best we can do for some constraints.

If an `all_different` constraint creates the direct encoding of x, and a `max` constraint creates subsequently the order encoding of x, then a channeling constraint from the new to the old encoding is required to make sure that information propagates from one to the other. If we omit channeling, then the assignment of the direct encoding will reflect the `all_different` constraint, and the assignment of the order constraint the `max` constraint, but neither is guaranteed to be fully correct. A channeling constraint will ensure either or both assignments to reflect both constraints.

So, for three encodings, we need three bi-directional[5] channeling constraints (one per pair of encodings). One simple way to express to create these is to post

[4] A cut-off limit (e.g., 128) that is compared to the domain size to decide between direct and binary or order and binary would be easy to add at this point.

[5] We have also developed some theory around one-way channeling constraints, but its implementation is not used in the final evaluation. The details can be found in appendix B.

a PB constraint which expresses the equality between the encodings, using the equation 12 to rewrite the value of each encoding into a linear expression:

$$\sum_{j \in D(x)} j f(x = j) = d_1 + \sum_{j \in D(x) \setminus \{D_l(x)\}} f(x \geq j) \tag{13}$$

Similar linear constraints can be constructed for the other two channeling between order and binary and between direct and binary. However, these channeling constraints are unnecessarily costly, so we will introduce better methods will be introduced in the next section.

4.1.4 Channeling constraints

We created six channeling constraints method, two per pair of encodings. All use no auxiliary variables and far fewer clauses than the above PB approach, because they use the underlying patterns of the encoding which is lost when the variable is pseudo-Booleanized. For example, the fact that $f(x, 0)$ eliminates all the even values of x so that $f(x \neq 1)$, $(x \neq 3)$, etc. is left for the pseudo-Boolean encoder and/or the SAT solver to figure out, while the our new channeling constraint make use of this fact directly which leads to much smaller encodings. In section 5.1.1, we will do small experiments to evaluate which channeling constraint methods to use.

Depending on the implementation (and certainly for MiniZinc), the domain of x might be pruned between the creation of two encodings. This is allowed as long as we consistently use the full set of encoding variables. From here onwards, we will use D, O and B to refer to the index sets of the direct, order and binary encoding arrays, respectively.

direct $\Leftrightarrow$ order, *direct* method [22] Trivial:

$$\bigwedge_{i \in D \cup O} f(x = i) \leftrightarrow f(x \geq i) \wedge f(x < i + 1) \tag{14}$$

direct $\Leftrightarrow$ order, *order* method This method might be equivalent to the former, even though it only uses negated direct encoding variables. Any order encoding variable $f(x \geq i)$ eliminates all values strictly less than i, that is, it implies $f(x \neq i - 1)$, $f(x \neq i - 2)$, etc.. This is symmetrical for $f(x < i)$, which eliminates values greater than (and including) i:

$$\bigwedge_{i \in D \cup O} ((f(x \geq i) \rightarrow f(x \neq i - 1)) \wedge (f(x < i) \rightarrow f(x \neq i))) \tag{15}$$

Notice that for $f(x \geq i)$ (and symmetrically for $f(x < i)$), we only need constrain $f(x \neq i - 1)$, but not $f(x \neq i - 2)$ (or $f(x \neq i - 3$, etc.), since the ordering constraint on the order encoding will take care of that through $f(x \geq i) \rightarrow f(x \geq i - 1) \rightarrow f(x \neq i - 2)$.

direct ⇔ binary, *binary* method [25, 30] For the following channeling constraints involving binary, having an example of a small two's complement encoding of $n = 3$ bits is helpful:

```
    j   j'
 i 012 012
==========
-4 001 000
-3 101 100
-2 011 010
-1 111 110
 0 000 001
 1 100 101
 2 010 011
 3 110 111
```

This table should be read as having seven columns: the left-most column denotes the assignment of $x = i$, and the next three columns show the corresponding assignment for each j-th bit. So, for example, for $f(i, j)$ with $i = -2$ and $j = 1$, we can see that the bit in the binary representation in row 3, column 3, has value 1 (true). The last three columns are the orderable binary encoding of the value of i, which is the same as the two's complement but with the sign bit ($j' = 2$) flipped (see formula 9).

Going back to the channeling constraint from direct to binary, we create implications from $f(x = i)$ which directly set the binary encoding variables according to the (two's complement) representation of i. Here we use the binary encoding function in a fixed context ($f_n(i, j)$), which, as shown in the table, returns the value of the j'th bit of the two's complement binary representation of i in n bits. We set n to match the number of bits of the binary encoding of x.

$$\bigwedge_{i \in D} \bigwedge_{j \in B} \begin{cases} f(x = i) \rightarrow f(x, i) & \text{if } f_{|B|}(i, j) \\ f(x = i) \rightarrow \neg f(x, i) & \text{otherwise} \end{cases} \tag{16}$$

For example, if $f(x = 2)$ and the binary encoding has $|B| = 3$ bits, then the two's complement of 2 is 010, so we post the implication $f(x = 2) \rightarrow \neg f(x, 0) \wedge f(x, 1) \wedge \neg f(x, 2)$.
[6].

direct ⇔ binary, *elimination* method As stated, we can generalize the fact that $f(x, 0)$ eliminates all the even values of x so that $f(x \neq 1)$, $(x \neq 3)$, etc.. The generalization is that if $f(x, j)$ is true, then i cannot be any value where $f(i, j)$ equals 0. For the first bit, $j = 0$ (the *even* case), this eliminates

[6] As noted in [25] and implemented in [30], the `at-most-one` constraint on the direct encoding may be omitted with this channeling constraint in place, as it will already be implied by the binary encoding. This was a late find, so in our project it is not implemented.

every other row, for $j = 1$ this eliminates every other *segment* of *two* rows $(f(x \neq -4), f(x \neq -3), f(x \neq 0), f(x \neq 1), ..)$, and for $j = m$ this eliminate every other segment of 2^m rows. Symmetrically, any false $f(x, j)$ eliminates any row i where the j'th binary column has a 1, so where $f(i, j) = 1$.

$$\bigwedge_{i \in D} \bigwedge_{j \in B} \begin{cases} \neg f(x, j) \rightarrow f(x \neq i) & \text{if } f_{|B|}(i, j) \\ f(x, j) \rightarrow \neg f(x, i) & \text{otherwise} \end{cases} \tag{17}$$

order $\Leftrightarrow$ binary, *subset* method For integer variable x, any true bit in its binary encoding with significance j individually contributes a value of 2^j to the value of x. This means that if $f(i, j)$ is true, then x has to be greater than the minimum value (the encoding's floor) plus 2^j, so for $n = |B|$ bits that would be: $f(x \geq -2^{n-1} + 2^j)$. Since the sign bit break this pattern by deducting its value, we can flip it and effectively use the orderable binary encoding function f' so that the last bit will behave like any other regular bit.

Looking at the table, we can see that a true first bit $f'(x, 0) = 1$ tell us that the first row (for $i = -4$) is no longer possible, which means $f'(x \geq -3)$. Likewise, a true second bit $f'(x, 1)$ will eliminate two rows ($i = -4$ and $i = -3$), so that $f'(x \geq -2)$. However, while the row for $i = -2$ is also eliminated by $f'(x, 0) = 1$, this first bit by itself is not enough to say $f'(x \geq -1)$, since x might equal -3. But, if *both* of the first two bits are true, only then is $f'(x \geq -1)$ indeed true. The developing pattern will cover the whole domain of x:

$$\begin{aligned} f'_3(x, 0) &\rightarrow f(x \geq -3) \\ f'_3(x, 1)) &\rightarrow f(x \geq -2) \\ f'_3(x, 0)) \wedge f'(x, 1) &\rightarrow f(x \geq -1) \\ f'_3(x, 2)) &\rightarrow f(x \geq 0) \\ &\dots \end{aligned} \tag{18}$$

Symmetrically, if $f'(i, j)$ is false, then even if all other bits are true and the sign bit false, the maximum value of x is pushed away from the ceiling by 2^j. E.g., if the second bit is false, then the last two rows are eliminated.

$$\begin{aligned} \neg f'_3(x, 0) &\rightarrow f(x < 3) \\ \neg f'_3(x, 1)) &\rightarrow f(x < 2) \\ \neg f'_3(x, 0)) \wedge \neg f'(x, 1) &\rightarrow f(x < 1) \\ \neg f'_3(x, 2)) &\rightarrow f(x < 0) \\ &\dots \end{aligned} \tag{19}$$

Generally:

$$\bigwedge_{i \in O} \left(\left(\bigwedge_{\{j \in B, f'(i,j)=1\}} f'(x,j) \rightarrow f(x \geq i) \right) \wedge \bigwedge_{\{j \in B, f'(i,j)=0\}} \neg f'(x,j) \rightarrow f(x < i+1) \right) \tag{20}$$

order ⇔ binary, *segment* method As we have seen in the previous channeling constraint, every binary column (at position j) consists of alternating segments of repeating 0's and 1's. For example, the column of $j = 1$ has four segments of size 2: `00110011`. The values of x can be in a specific segment (say, a x of -2 or -1 falls in the second segment), before a segment (x is -4 or -3) or after a segment (x is at least 0). There are no other possibilities.

Generally, with $n = |B|$ bits, the segments in column j will have a size of 2^j. Thus, there are $\frac{2^n}{2^j} = 2^{n-j}$ segments, which we will index with $k = 0, 1, \ldots, 2^{n-j} - 1$. Given that the floor value (the smallest value the encoding can encode) for n bits is 2^{n-1}), the k'th segment in column j starts at row with $i = s_{k,j} = -2^{n-1} + k2^j$. Segments with an even k have 0's and segments with an odd k have 1's, which we'll call 0- and 1-segments, respectively.

These segments are helpful, because when a certain bit $f(x, j)$ is true, x must be in one of the 1-segments, if $f(x, j)$ is false, it must be in one of the 0-segments. If we look at the row where $i = -2$, the corresponding binary table variable in column 2 is in the 2nd segment ($s_{2,2}$) contains 1's. This tells us that if $f(x \geq -2)$ is true, then two possibilities exist. Either $f(x, 2)$ is true (and x could be -2 or -1, or in any other 1-segment further down the table). Or, if $f(x, 2)$ is false, then x cannot be in the current 1-segment and so must be in the previous 0-segment ($f(x < -2)$) or in any later 0-segment ($f(x >= 0)$).

So, every row i, column j, corresponds to segment k. The value of x must be either *before* the current segment ($f(x < s_{j,k})$), *after* the current segment ($f(x \geq s_{j,k})$), or the bit $f(x, b)$ must match the value of segment k, either 0 or 1, so $\neg f(x, b)$ or $f(x, b)$, respectively. This last fact can be written as $f'(x, b) \equiv 1 - k \mod 2$. This leads us to the following constraint:

$$\bigwedge_{0 \leq j \leq n-1} \bigwedge_{0 \leq k \leq 2^{n-j}-1} f'(x,b) \equiv 1 - k \mod 2 \vee f(x < s_{j,k}) \vee f(x \geq s_{j,k+1}) \tag{21}$$

4.2 The `sat` redefinition library

The MiniZinc standard library (`std`) defines default flattening behavior. After standard compiler optimizations are done, only the predicates listed as FlatZinc built-ins are left for the solver. Global constraints are also decomposed according to their definition by `std`. We will need to redefine most built-ins and globals so that they are not left as-is but are expressed as CNF using the clause constraint `bool_clause`, so we can feed this to the (Max-)SAT solver.

If a solver has native implementations for global constraints, or if one wishes to implement custom decompositions for globals in MiniZinc, the option exists to provide a custom redefinition library that overwrites any `std` predicates (or prevent redefinition to leave the constraint for the solver). In our case, we add the `sat` redefinition library to redefine almost all FlatZinc built-ins and globals in order to end up with FlatZinc that is very close to CNF, which we call CNFlatZinc. It consists of Boolean variables, arrays of Boolean variables, `bool_clause` constraints, `bool_not` constraints, cardinality and linear constraints for PBLib and (in case of optimization) an objective value given as a linear expression. If some integer variables are unconstrained, they are left as-is and resolved in the interface as well. We resolve everything to CNF after the `sat` library, in the SAT solver instance (see 4.3).

A predicate definition does not have to be purely MiniZinc. New functions can be registered as C++ built-ins instead. This is useful, for example when performance is important, when integrating with existing C++ projects, when something is easier to express using the imperative paradigm, when C++ tooling is required, and so on.

Every predicate C implicitly come with a `*_reif` version and a Boolean variable r version in which $r \Leftrightarrow C$. This is needed frequently when compound constraints are flattened, which replaces sub-constraints by their reification r. If no `*_reif` version is defined, the compiler generates a default decomposition, but in some cases it is beneficial to make a special redefinition if we have a better decomposition than the default. For some constraints, we can handle the non-reified and the reified constraint at the same time by fixing $r = 1$ for the non-reified constraint. If in this section a reified version of a built-in is missing, it means we leave it to be auto-generated.

To get an idea for how FlatZinc built-ins are redefined in the `sat` library, let's look at an example. The MiniZinc compiler will generate a FlatZinc built-in predicate `bool_and` for $p \wedge q \Leftrightarrow r$ can be normalized to $p \vee \overline{r} \wedge q \vee \overline{r} \wedge r \vee \overline{p} \vee \overline{q}$. The `bool_clause` predicate has two arguments, one array of Boolean variables for the positive literals and one for the negated literals, so the our redefined decomposition of `bool_and` into CNFlatZinc is:

```
predicate bool_and(var bool: p, var bool: q, var bool:
  r) =
  bool_clause([p],[r]) /\ bool_clause([q],[r]) /\
    bool_clause([r],[p,q]);
```

For this built-in, all arguments are variable (as indicated by the keyword `var`). The MiniZinc compiler can overload functions on its arguments number, order and/or type, so another version of `bool_and` can be provided with `r` as parameter with `[..], var bool: q, bool: r)`.

4.2.1 Dynamic encodings in MiniZinc

Before we go through the redefinitions, we show here how the dynamic encoding theory from section 4.1.3 is implemented in MiniZinc. The goal is to make every

constraint fully dynamic, so that we can always re-use an existing encoding and are never forced to re-encode. If this is realized, then `mzn-sat` can be configured to exclusively use a single encoding (e.g., order encoding mode), in addition to a *mixed* mode in which it might mix multiple and do re-encodings.

At the core, the library employs encoding creation functions such as `direct_encode(var int: x)` which all execute the following four steps:

1. Instantiate the Boolean encoding variables for x.
2. Post any necessary constraints on the Boolean encoding variables (such as the exactly one constraint in the case of `direct_encode`).
3. Either:
   (a) If x has no pre-existing encodings, it will apply a *reverse mapper* to x, which tells the compiler how the assignment of the Boolean encoding variables can be mapped back to the value of x. At this point, x is no longer needed, so it is moved to the *output model* so that in the output phase, the solver's solution can be assigned to x.
   (b) If there is a pre-existing encoding, then a reverse mapper for that encoding should already be in place. Instead, we create either a one-way channeling constraint (from the new to the pre-existing encoding), a two-way channeling (using two one-way channeling constraints) or a PB-based two-way channeling constraint, depending on a user option.
4. Finally, we annotate x with an encoding annotation (`::direct_encoded`). At various locations, such as in the previous step, the library can inspect which variable has which encoding annotation using the new built-in, `has_ann`. If no encoding annotation is present, then the encoding function knows it is producing the first encoding, which is designated as the one and only reverse mapper to x (the mechanism that maps the assignment of the encoding to the assignment of x).

Which encoding creation function is used for which integer variable is decided during the flattening process in the `sat` library redefinitions. For instance, the redefinition for `int_le(var int: a, var int: b)` (for the inequality constraint $a \leq b$ between integer variable a and fixed integer b) will generally dispatch an *encoding specific* redefinition (for instance, `int_le_order`). Since an inequality constraint works well with the order encoding, this is what it will try to use in most cases. Here is a slightly simplified version of the redefinition:

```
predicate int_le(var int: a, var int: b) =
  if ub(a) < lb(b) then true
  elseif lb(a) > ub(b) then false
  else
    if has_ann(a, order_encoded) /\ has_ann(a,
      order_encoded) then
      int_le_order(a, b)
```

```
  elseif has_ann(a, binary_encoded) /\ has_ann(a,
    binary_encoded) then
     int_le_binary(a, b)
  elseif has_ann(a, direct_encoded) /\ has_ann(a,
    direct_encoded) then
     int_le_direct(a, b)
  else
       annotate_sat_encoding(a,
         SAT_INT_INEQUALITY_DEFAULT_ENCODING)
    /\ annotate_sat_encoding(b,
      SAT_INT_INEQUALITY_DEFAULT_ENCODING)
    /\ int_le(a, b)
  endif
endif;
```

First, some trivial checks are done which might decide the constraint without the need for an encoding, which has the added benefit that the encoding specific redefinitions do not have to account for these edge cases. In the following formal redefinitions, we won't show all the trivial checks for the sake of readability. Then, the encoding annotations (attached to the integer variable by the encoding creating function, e.g. `order_encode(var int: x)`) are checked on the integer variable to see if the existing encodings can be re-used for any of the encoding specific redefinitions. In the `int_le` case, we first check for the order encoding since that is the preferred choice for an inequality constraint [22].

If no suitable existing encodings exists, the `annotate_sat_encoding` will add the redefinition's preferred encoding annotations (set in the second argument as `SAT_INT_INEQUALITY_DEFAULT_ENCODING`, which by default is set to `order_encoded`) and call `int_le` recursively with the annotated variables. Another library option exists for the user to override the constraint's annotation of the second argument so that `annotate_sat_encoding` (and thus the entire library) will always encode with one specific encoding for the whole model. In fact, in this case we assert that the other encoding creation functions are not used.

This implementation has the following consequences:

- Re-encodings are somewhat rare, since an existing encoding will always be re-used if possible. For `int_le`, a re-encoding only happens if one variable has one encoding, and the other has either no encoding or a different one. This is an ok starting point, because while a re-encoding might be beneficial in specific cases, we expect re-use to be generally better. To leverage re-encodings in cases where re-use is also possible, the library will need some new building block to make intelligent decisions.

- In MiniZinc, there is no formal guarantee of the flattening order. Yet, the flattening order of the constraint in the user model now becomes important, as *earlier* constraints (i.e., constraints that are flattened first) get to make the first choice on the preferred encoding. For example, if an

earlier constraint on integer variable has a weak preference for direct over order, and the later constraint only supports order, the re-encoding and channeling from order to direct encoding could have been avoided if the order of constraints were reversed. An example would be `int_eq(x, y) /\ int_le(x, z)`, which would re-encode `x`.

- Since the user can also add annotations in their model, they can steer what encodings are used on the variable level. This allows for quite granular control, which has proven usefull in development, testing an earlier benchmarking.

- Higher-level redefinitions use lower-level predicates, which defer the encoding decisions to the redefinitions of the lower-level predicates. For instance in this (simplified) `array_int_maximum` redefinition:

```
predicate array_int_maximum(var int: m, array[int]
  of var int: xs) =
  forall(x in xs)(x <= m) /\ let { var
    index_set(xs): i; } in m == xs[i];
```

  The use of <= and == will generate `int_le` and `int_eq` constraints which will decide the actual encoding. Note that when $x = v$, $x \geq v$, etc. is used in the formal definitions, these are to be read as equality or inequality constraints, not as the encoding variables of any particular encoding, $f(a = v)$ or $f(a \geq v)$, respectively.

- Constraints (that do not defer their encoding choice) have two subtly different ways to specify their order of preference. The first is which encoding is preferred to be re-used (if multiple encodings exist), which is decided by the order of cases in the `if`-statement. Currently, this is cumbersome to change (for the user). The second is which encoding will be created if none exists, which can be flexibly changed for a whole category of redefinitions in the library's option, for which the defaults are:

```
% Constrain default encodings
SAT_INT_ABS_DEFAULT_ENCODING = SAT_BINARY;
SAT_INT_DIV_DEFAULT_ENCODING = SAT_BINARY;
SAT_INT_EQUALITY_DEFAULT_ENCODING = SAT_DIRECT; %
  also for int_lin_eq, int_ne
SAT_INT_INEQUALITY_DEFAULT_ENCODING = SAT_ORDER; %
  also for int_lin_le, min/max
SAT_INT_PLUS_DEFAULT_ENCODING = SAT_BINARY;
SAT_INT_MOD_DEFAULT_ENCODING = SAT_BINARY;
SAT_INT_TIMES_DEFAULT_ENCODING = SAT_BINARY;
SAT_SET_IN_DEFAULT_ENCODING = SAT_ORDER;
SAT_OBJ_INT_LIN_EQ_DEFAULT_ENCODING = SAT_DIRECT;
```

By default, the first preference (in the first if-statement) for which encoding type is re-used is the same as the preference for which encoding is created.

4.2.2 FlatZinc built-ins: Boolean

The same simple transformations we have seen for `bool_and` can be done for `array_bool_and`, `array_bool_or`, `bool_eq`, `bool_eq_reif`, `bool_le`, `bool_le_reif`, `bool_lt`, `bool_lt_reif`, `bool_ne_reif`, `bool_or`, `bool_xor` and `bool_clause_reif`. More interesting constraints follow, which we indicate here compactly using regular expression-like optional substrings and disjunction (`[..]`, `|`, etc.).

`array_bool_[or|and](array [int] of var bool:` A`, [var] bool:` r**`)** Constrain $r \Leftrightarrow \bigwedge A$ or $r \Leftrightarrow \bigvee A$, of which the general CNF is trivial. We also handle edge cases where A (after removing fixed elements) has 0 or 1 variables, so that we can fix r or unify it with the single element, respectively.

`array_bool_element`(var int: b**, array[int] of bool:** A**, var bool:** c**)** Constrains $A_b \Leftrightarrow c$. Like for other element constraints, we defer these to set constraints (see 4.2.3) by constraining the result c to be equivalent to whether index b points to one of the true elements of A. Here $I(A)$ returns the index set of A, the interval from the first and last index of A. As mentioned, MiniZinc supports variable index sets, so these are not necessarily 0- or 1-indexed as in other programming languages.

$$(b \in \{i : i \in I(A), A_i\}) \Leftrightarrow c \tag{22}$$

`array_var_bool_element(var int:` b`, array [int] of var bool:` A**`, var bool:` c**`)`** Same as above, but with A containing variable Booleans instead. Instead of a bidirectional implication, we just use a one-way implication from index b to result c. We defer this to lower level built-ins like `(b == i)`. Since it is part of a compound expression, this equality constraint will be reified using `int_eq_reif(b, i, r)` (see section 4.2.3), which will then likely use the direct encoding. This shows how close (syntactically and semantically) an expression like `(b == i)` is to the direct encoding function function $f(b = i)$ (or `b >= i` would be to the order encoding function $f(b \geq i)$).

```
predicate array_var_bool_element(var int: b, array[int]
  of var bool: as, var bool: c) =
  forall(i in dom(b))(
    (b == i) -> (c == as[i])
  );
```

bool_lin_[le|eq](array [int] of int: A, array [int] of var bool: B, [var] int: c) Constrains an n-term PB constraint $\sum_{i=0}^{n} a_i x_i \ \# \ c$ for comparators $\# \in \{\leq, =\}$, respectively. If c is non-fixed, we encode it as a PB linear expression (the same as any integer linear term, see equation 12 in section 4.1.3) so we get a PB constraint with a fixed c. This is left as a `pblib_bool_lin_le` or `pblib_bool_lin_eq` constraint in the CNFlatZinc.

bool_lin_[le|eq]_reif(array [int] of int: A, array [int] of var bool: B, int: c, var bool: r) The reified version of the PB constraint $C_{\#}$, namely $r \Leftrightarrow C_{\#}$. PBLib supports half-reifications through all of its encoding methods. With the following method (taken from MiniZinc's `linear` library [11]), we can constrain full, bi-directional reification using multiple half-reified, conditional PB expressions and an auxiliary variable p:

$$(r \Rightarrow C_{\geq}) \wedge (p \Rightarrow C_{<}) \wedge (r \neq p) \tag{23}$$

We don't have a specific version for comparator $<$, but we can easily convert to it with $C_{\leq}$ by using $c - 1$. Here, r depends bi-directionally on $C_{\geq}$, because we also handle the case where the constraint is false: $\overline{C_{\geq}} \Leftrightarrow C_{<}$.

Similarly, for equality PB constraints we create two auxiliary variables, p and q, again constraining $C_{=}$, and $\overline{C_{=}}$ with two inequality constraints:

$$(r \Rightarrow C_{=}) \wedge (p \Rightarrow C_{<}) \wedge (q \Rightarrow C_{>}) \wedge (r \Leftrightarrow (\overline{p} \wedge \overline{q})) \tag{24}$$

bool_*_imp In addition to reification such as `bool_and` ($r \Leftrightarrow p \wedge q$), a redefinition library also allows us to provide half-reification predicates such as `bool_and_imp` ($r \Rightarrow p \wedge q$) which in some cases can be used to flatten compound constraints [31]. Since a half-reification requires only one implication direction, it yields generally smaller, more efficient encodings: $\overline{p} \vee \overline{q} \vee r$ in this case. However, they cannot be used in all reification scenarios, so we still require the regular, bidirectional reified constraints. The FlatZinc Boolean half-reifications have been implemented as well, but not the FlatZinc Integer half-reification due to time constraints.

4.2.3 FlatZinc built-ins: integers[7]

int_eq[_reif]([var] int: a, [var] int: b[, [var] bool: r]) Constrains $a = b$, possibly reified in r. This constraint is only generated when reified, or if either a or b has an existing encoding. Otherwise, the integer variables can be unified with each other.

If either of the variables (say, b) is fixed, then we can fix all the encoding variables of a. Since this will remove variables from the model, we do it for *every* existing encoding. From now on, we indicate that all cases apply in the

[7] The *arithmetical* FlatZinc built-ins (`int_plus`, `int_times` and `int_div` are implemented as well, but are not evaluated as no constraints the benchmarks models (from the MiniZinc Challenge 2019) do not include these constraints). Their details are in appendix G

following equations by using $\bigwedge$ (instead of the conventional interpretation where only one case applies).

$$\bigwedge \begin{cases} f(a = b) \wedge \bigwedge_{i \in D \setminus \{b\}} f(a \neq i) & \text{direct} \\ \bigwedge_{O_l \leq i \leq b} f(a \geq i) \wedge \bigwedge_{b+1 \leq i \leq O_u} f(a < i) & \text{order} \\ \bigwedge_{i \in 0..n-1} f(a, i) = f_n(b, i) & \text{binary} \end{cases} \tag{25}$$

As mentioned, the compiler can generate reified versions of constraints, but for the direct and order encoding these would be needlessly large, so we explicitly provide these reified versions:

$$\begin{cases} r \Leftrightarrow f(a = b) & \text{direct} \\ r \Leftrightarrow f(a \geq b) \wedge f(a < b + 1) & \text{order} \\ r \Leftrightarrow a = b & \text{binary (auto-generated)} \end{cases} \tag{26}$$

Since in this case r is equivalent to the direct encoding variable that would be generated to represent $f(a = b)$, we would like to re-use r for a potential later direct encoding even if we are not using the direct encoding on this constraint. We added an option to the `sat` library that will enable what we call **pre-emptive partial encoding**, where for `int_eq_reif` we create a direct encoding variable $f(a = b)$ *even* if we are encoding with order or binary and unify it with r. Then, if a direct encoding is created for a later on, then calling the encoding variable creation function on a for value b will return r. This is also implemented for `int_le_reif`. This behaviour is provided as a library option, turned off by default since it is experimental.

If both variables are non-fixed, then we can still unify the Boolean variables (that have the same index) from either encoding with each other if the encodings are the same, which given the definition of the encoding function will work even if indices of the variables from one encoding are out of bounds of the other encoding (e.g., if $i > D_u(b)$, then for direct encoding $f(a = i) = f(b = i) = false$). If the encodings are different, we can use channeling constraints and still re-use the encoding.

`int_le([var] int:` a`, [var] int:` b`)` Constrains $a \leq b$. This constraint is only generated when either a or b has an existing encoding, otherwise the variables the domain of a/b is shrunk directly.

$$\left(\bigwedge\right) \begin{cases} \bigwedge_{i \in b+1 \leq i \leq D_u} f(a \neq i) & \text{direct} \\ \bigwedge_{b+1 \leq i \leq O_u} f(a < i) & \text{order} \\ (f'(a, n-1), f'(a, n-2), \ldots, f'(a, 1), f'(a, 0)) \geq_{\text{lex}} & \\ (f'(b, n-1), f'(b, n-2), \ldots, f'(b, 1), f'(b, 0)) & \text{binary} \end{cases} \tag{27}$$

If either variable is fixed, all cases apply, otherwise only the preferred encoding (order) applies. This redefinition (like `int_eq`) supports the pre-emptive partial encoding feature, but this time for the order encoding: $r \Leftrightarrow f(a, b)$ if a is fixed or $r \Leftrightarrow f(a \leq b) \Leftrightarrow \neg f(a \geq b + 1)$ if b is fixed.

`int_ne([var] int:` a`, [var] int:` b`)` If one of the variables is fixed (say, b is), we can simply fix direct encoding variables to false or unify neighbouring order encoding variables (since we already have $f(a \geq b{+}1) \rightarrow f(a \geq b)$ from the order constraint), which in both cases will remove a Boolean variable from the model. Only for the binary encoding we will have to create a new clause:

$$\bigwedge \begin{cases} f(a \neq b) & \text{direct} \\ f(a \geq b) \leftrightarrow f(a \geq b+1) & \text{order} \\ \bigvee_{i \in B} f(a,i) \neq f(b,i) & \text{binary} \end{cases} \tag{28}$$

If both are variables, we use the reified version with a fixed, false $r = 0$ for direct or order encoding, or post a PB constraint $a - b = 0$ if the encodings are in binary.

`set_in(var int:` x`, set of int:` S`)` Constrains $x \in S$. We defer to negative equality constraints (`int_ne`) to eliminate values not in S:

$$x \in S \Leftrightarrow \bigwedge_{v \in D(x) \setminus S} x \neq v \tag{29}$$

`set_in_reif(var int:` x`, set of int:` S`, var bool: r)` Constrains $x \in S \Leftrightarrow r$. When considering an example set $S = \{1, 2, 3, 8, 9, 10\}$, there are two approaches to constraining x to be equal to one of the elements of S. The first is to use create a disjunction of *equality* constraints, which would favour the direct encoding:

$$\begin{aligned} r &\Leftrightarrow \bigvee_{v \in S} x = v \\ &\Leftrightarrow (x = 1 \vee x = 2 \vee x = 3 \vee x = 8 \vee \ldots) \end{aligned} \tag{30}$$

The second approach [8] is to use *inequality* constraints on every contiguous *range* that is present in S. As noted in section 4.1.1, a set S may be converted (and is internally stored as) an array R of (non-empty) intervals (ranges), $R = ([1,3], [8,10])$. These ranges offer us the opportunity to instead use inequality constraints, which for the order encoding only requires four clauses per range – no matter the range's size.

$$\begin{aligned} r &\Leftrightarrow \bigvee_{v \in S} x = v \\ &\Leftrightarrow \bigvee_{[l,u] \in R} x \geq l \wedge x \leq u \\ &\Leftrightarrow ((x \geq 1 \wedge x \leq 3) \vee (x \geq 8 \wedge x \leq 10)) \end{aligned} \tag{31}$$

[8] On the morning of the thesis defence, this second approach was noted to be similar to what is called Unified Domain Refinement in the work published on the MiniZinc linearization library [11]

As a range grows, the number of reified equality constraints it requires increases but the number of reified inequality constraints stays constant. Since a set with fewer gaps will convert to larger (and fewer) ranges, it should benefit the order encoding over the direct encoding. A heuristic based on the set composition is needed to decide between the direct and the order encoding, which commits us to the first, equality approach or the second, inequality approach for the whole set.

For the binary encoding, we can switch between equality and inequality constraints within the same set (from one range to the next). Since inequality constraints use lexicographical constraints (from `int_le_binary`) and equality constraints use logarithmically sized single clauses (from `int_ne_binary`), it will depend on the range's size, but perhaps also on the number of bits, when two inequality constraints will outperform the multiple negative equality constraints. For example, it might be cheaper for a range of size 5 to be constrained with 5 negative equality constraints instead of 2 inequality constraints, but not for a range of 6. After some small experiments, we use equality constraints for ranges up to size 7 when using the binary encoding:

$$\bigvee_{[l,u]\in R} \begin{cases} \bigvee_{l\leq i\leq u} x = i & \text{if } u - l + 1 \leq 7 \\ x \geq l \wedge x \leq u & \texttt{otherwise} \end{cases} \tag{32}$$

This warrants further research, especially regarding the heuristic. Since we let most `element` constraints defer to the reified set redefinitions, this constraint is actually very important. In section 5.1.2, we experimentally evaluate the effects of sets with different range sizes on the literal and clause counts for each encoding.

array_int_element(var int: b, array[int] of int: A, var int: c) Constrains $A_b = c$. There is a two way relationship between the index variable b and the result variable c, since a known b will fix the result directly ($b = v \rightarrow c = A_v$) and a known c will fix b to be in the set of indices $G = \{G_1, G_2, \ldots\}$ for which $A_{G_1} = A_{G_2} = \cdots = c$. The new `group` built-in creates an array of these non-empty sets (groups) of indices for each distinct value in A. In other words, `group` on A returns an array of sets of indices that point to the same value in A. The `group` built-in creates groups efficiently by employing an *arg sort* before grouping.

For instance, `group([2,2,1,1,1,2]` will return an array of two sets, `[3..5,{1,2,6}]`, because the 2's are in positions 1, 2 and 6 and the 1's are in positions 3 to 5 (automatically represented by MiniZinc as a range). Now, the value of c simply depends on the index associated with the set that b is in. We can clearly see that the number and size of the ranges with the groups depend on how many neighbouring elements of the same value occur in A.

If A is entirely composed of unique elements, then every set will have size 1 since each element will have a unique element. In that special case, assignments of b and c are permuted by A, so we can skip the `group` built-in.

$$\begin{cases} \bigwedge_{i \in I(A)} b = i \Leftrightarrow c = A_i & \text{if } |D(A)| = |A| \\ \bigwedge_{G \in \texttt{group}(A)} ((b \in G) \Leftrightarrow f(c = A_{G_1})) & \text{otherwise} \end{cases} \tag{33}$$

`array_var_int_element(var int:` b`, array [int] of var int` A`, var int` c`)` Same as above, but with A containing variable integers instead. The redefinition is almost exactly the same as for `array_var_bool_element` in section 4.2.2, as we use a one-way implication from index result: $b = v \rightarrow c = A_v$.

`array_int_maximum(var int:` m`, array[int] of var int:` X`)` Constrains $m = \max(X)$ (or $m = \min(X)$) for integer variable m and array of integer n variables X. Such constraint are naturally constrained using order encoding, for the other encodings we defer to inequality constraints and an element constraint on a new variable i with a domain equal to the index set of X:

$$\begin{cases} \bigwedge_{d_i \in D(m)} \left(f(m \geq d_i) \Leftrightarrow \bigvee_{j=0}^{n} f(X_j \geq d_i) \right) & \text{order} \\ \left(\bigwedge_{x \in X} x \leq m \right) \wedge m = X_i & \text{direct or binary} \end{cases} \tag{34}$$

`int_abs(var int:` a`, var int:` b`)` Constrains b to be the absolute value of a.

For a direct encoding, D is the direct encoding array index set, D_l is the minimum index and so $\max\{1, D_l\}$ is the minimal positive index. The trick is that there are now two ways for b to be equal to some value i, namely through $a = i$ or $a = -i$:

$$\begin{aligned} & \bigwedge_{\max\{1, D_l\} \leq i \leq D_u} f(b = i) \Leftrightarrow (f(a = i) \vee f(a = -i)) \\ \wedge & \bigwedge_{D_l \leq i \leq \min\{-1, D_u\}} f(b \neq i) \\ \wedge & \, (f(a = 0) \Leftrightarrow f(b = 0)) \end{aligned} \tag{35}$$

For order encoding, the value of b depends on whether a is more negative or more positive:

$$\begin{aligned} & \bigwedge_{D_l \leq i \leq 0} f(b \geq i) \\ \wedge & \bigwedge_{0 \leq i \leq D_u} (f(a \geq i) \vee f(a < -i + 1)) \Leftrightarrow f(b \geq i) \end{aligned} \tag{36}$$

For the binary encoding, we use the property of the two's complement scheme that flipping the sign bit and adding one to the binary encoding of x represents $-x$, so if a has n bits:

$$(a = b) \vee (((f'(a,0), f'(a,1), \ldots) + 1) = (f(b,0), f(b,1), \ldots)) \qquad (37)$$

`int_lin_[le|eq|ne]` Constrains an n-term IP constraint $\sum_{i=0}^{n} a_i x_i \ \# \ c$ for comparators $\# \in \{\leq, =, \geq\}$, respectively, and its reified versions with Boolean variable r. We use the dynamic encoding to transform the IP constraint to PB as described in section 4.1.3, and pass it on to the matching `bool_lin_[le|eq]`.

For `int_lin_ne`, we create a auto-generated reified linear constraint with $r = 0$, which will use `bool_lin_[eq|le]_reif` from section 4.2.2. For reified version, the 0 and 1 terminals are set to $\overline{r}$ and r, respectively. It should be noted that only in this case (and for the objective function, see section 4.2.6) dynamic encoding is used, as in all other cases either direct or order encoding is forced. Depending on the model, this still accounts for a big chunk of the constraints, but ideally every case (and every other constraint) would support dynamic encodings.

4.2.4 FlatZinc built-ins: sets

Apart from FlatZinc built-ins for Boolean and integer constraints, there are also the set constraints (other than the already discussed `set_in[_reif]`). For the most part, the `nosets` library takes care of this for us by redefining (almost) all set constraints to Boolean/integer constraints. To do this, it transforms a set of (variable) integers S to arrays of (variable) Booleans $B = (b_i : D_l(x) \leq i \leq D_u(x))$ that are a bit string representation of S with an `set2bools` encoding function $f(v \in S) = b_v$ such that $v_b \Leftrightarrow v \in S$. This is similar to the direct encoding of a single integer, but as there can be multiple different values in a set, there is no exactly-one constraint.

4.2.5 Global constraints

Some globals have been implemented for the `sat` library, but most are either left as their `std` decomposition (which in some cases is a solid choice too), or still have an implementation copied from the `linear` library (these are untested for our `sat` library and are not used in any of the benchmarks).

`all_different, all_different_except_0` Constrain an array of integer variable to be pair-wise different. As we have seen in the Sudoku example in section 2.3, the `all_different` constraint can be encoded by posting an `at_most_one` on every column of the encoding grid made up by the direct encodings of the input array of integer variables. For `all_different_except_0`, we do the same except we don't constrain the 0'th column.

`count` For the integer cardinality constraint, MiniZinc has the `count` global for an array of integer variables X, an integer variable for the count c, and an variables integer for y, which is the element to be counted. So, if y is non-fixed,

we don't even know *what* element we're counting yet, if c is non-fixed we don't know how *many* of those y's there can be.

Because different solvers might or might not support different scenarios, there are about 30 versions of the `count` global: one for every conceivable comparator (`count_eq` ($=$), `count_ne` ($\neq$), `count_le` ($\leq$), `count_lt` ($<$), etc..), for each of those a version for every comparator where c and y are fixed and one where they are not, and for each of those a reified version as well. In the `std` library, the redefinitions all lead back to the redefinition of `count_eq`.

Since we have specializations for minimum, maximum and exact cardinality in PBLib, we funnel the comparators down to $\# \in \{\leq, \geq, =\}$, respectively (e.g., `count_lt` can be redirected to `count_le`). Then, if c is fixed, we convert the integer variable to Boolean variables $Y = \{x = y : x \in X\}$, which we can encode using PBLib (e.g., by posting a `pblib_bool_count_geq`). We handle the following trivial cases in the library (assuming $\geq$-comparators unless otherwise noted, since the others are similar). Some of these are handled in PBLib also, but this allows us to fix variables early which propagates throughout the flattening phase:

- Every fixed $y \in Y$ are removed and each decrements c by one
- If $c < 0$, the constraint is trivially unsatisfiable.
- If $c = 0$, then all y are false
- If $c = 1$ and $|Y| = 2$, then $\overline{y_1} \vee \overline{y_2}$
- If $c = 1$, $|Y| = 2$ and $\#$ is the $=$-comparator, then $y_1 \neq y_2$

For reified versions or if c is non-fixed, we create a linear expression of $(r \Leftrightarrow) \sum Y \# c$, representing Y using integer variables with domain $\{0, 1\}$. An alternative approach for non-fixed c's uses (our own implementation of) sorting networks, but PBLib needs to be adapted before it can be used. The approach is shown in appendix H.

4.2.6 Optimization

To handle optimization, the idea is to encode the objective into the weights of the soft clauses of the model. If the objective function is flattened down to its result, a single integer variable x, we can encode it with (for instance) in the direct encoding, and add each variable v_{d_i} as a soft unit clause with weight d_i, which we will represent here as the tuple (v_{d_i}, d_i). The solver will prioritize the weights correctly, and the objective value can be calculated from the solution cost (the sum of unsatisfied clauses) that is output by our Max-SAT solvers for (intermediate) solutions. Since our Max-SAT solvers don't output intermediate solutions (unless interrupted by a timeout or by the user), we have to rely on this cost value to monitor the optimization progress. This calculation and other aspects of the optimization are handled by the interface (see section 4.3.1).

However, if the objective function is a complicated expression, then the relationship of x to the variables in that expression will be hard to propagate on for the solver. So, the method was improved by instead flattening the objective function down not to a single integer variable, but only to a linear expression $c + \sum_{i=0}^{n} a_i x_i$, if possible[9], which is then handled in the special `obj_lin_exp` constraint. We use the dynamic encoding for linear expressions (see section 4.1.3, formula 12) to convert the linear expression to a PB linear expression $d + \sum_{j=0}^{m} b_i y_i$ of some size m. Then, we create soft unit clauses for each $0 \geq j \geq m$ terms, if b_j is non-fixed and a_j is non-zero. Since the target MaxSAT format (WDIMACs) requires weights to be positive, we use the following trick to guarantee that all non-fixed soft clauses have positive weights:

$$\begin{cases} b_j y_j = (b_j, y_j) & \text{if } b_j > 0 \\ b_j y_j = b_j(1 - \overline{y_j}) = b_j - b_j \overline{y_j} = (1, b_j) \wedge (\overline{y_j}, -b_j) & \text{if } b_j < 0 \end{cases} \tag{38}$$

The fixed soft clauses ($(1, b_j)$) can be aggregated as sum into one fixed soft clause, along with any other fixed terms from the PB linear expression, and the d term. This fixed soft clause might have a negative weight, but it will not be included in the model. Instead, it is just added to the objective value by the SAT interface during output.

Like any other dynamic constraint, the `obj_lin_exp` constraint supports all encodings – even if they are mixed. In this case we don't know if one is better than the other. While binary would create the fewest soft clauses, the difference in flattening time will be much smaller than when constructing a BDD for a linear constraint. As to propagation strength, we don't have any speculation yet (and it might be solver dependant, or not make much difference at all). This is why the purpose of the dynamic encoding here is to re-use the choice of encodings of earlier constraints that do have a preference. For this reason, we want to implement `obj_lin_exp` in such a way that it is flattened as late as possible, when many or all encodings have been decided by constraints that benefit more by having the choice.

4.2.7 Testing

Most of the interface, and the library in particular, has been unit tested using the (slightly adapted) unit test suite of MiniZinc in which (all) expected solutions can be provided to a model. We use variables with small domains and check all solutions against a spec that is easily generated as test specification using any other solver back-end. With this, the correctness of any given predicate can be established with high confidence (and for smaller constraints, the whole input space can be tested). Around a hundred of such tests have been added.

[9] This aggregation was implemented by my co-supervisor.

4.3 The SAT solver interface

The flattened model (CNFlatZinc) is given to the SAT solver interface, which handles interaction between SAT solvers and the MiniZinc process and takes the last steps that converts CNFlatZinc to CNF readable by the solver. Unfortunately, the behavior of (Max-)SAT solvers is not always consistent, so some assumptions that are made could make the interface unsuitable for some solvers [10]

4.3.1 From CNFlatZinc to (W)DIMACS

As SAT solver input format, the interface supports only the DIMACS format as specified by the SAT competition, to which most SAT solvers adhere. The following DIMACS represents $p \vee \overline{r} \wedge q \vee \overline{r} \wedge r \vee \overline{p} \vee \overline{q}$:

```
c My CNF model
p cnf 3 3
1 -3 0
2 -3 0
3 -1 -2 0
```

The `c`-lines are comments and the `p`-line lists the format (`cnf//wcnf`), number of variables and of clauses. The other lines are clauses wtih the literals expressed numbered DIMACS IDs from 1 to n, which are negative if negated. The p, q and r variables of our example formula have been mapped by the interface to 1, 2 and 3, respectively. To build the DIMACS file from CNFlatZinc, and to later parse the solution assignment values back to the MiniZinc output model for the user, we construct an *internal model* that maintains an efficient (constant look-up) two-way `map` between FlatZinc variables and DIMACS IDs, and a clause list.

First, we iterate over the `bool_not` constraints, as its two arguments p and q can share the same DIMACS variable if we store them as `map(p)=-map(q)`. For `bool_clause` constraint, we resolve fixed variables (which could decide the model at that point in compilation if we end up with an empty clause). The variables are added to the map, which generates new DIMACS IDs lazily.

Cardinality and PB constraints are also still present in the CNFlatZinc, for which the SAT interface will use our fork of the open-source library PBLib [33] to encode the PB constraint into CNF. This library supports different BDDs, bimander, sorting networks and other methods, and uses a quick analysis to choose whatever method is estimated to require the lowest number of clauses. In our adaptation, we made some usability improvements, and added experimental support for reified PB constraints by setting the reification variable BDD terminal nodes equal to the terminal nodes, which is how our original BDD implementation worked (see appendix F). It was not stable enough to use in the evaluation, so instead we used the method described in section 4.2.2

[10] Although the recent SAT heritage project [32] aims to provide a consistent interface for every SAT solver. This should be added in the future to the MiniZinc-SAT interface, but as Max-SAT solvers were our primary interest this was not a priority to add.

which uses multiple half-reified constraints, which was already supported in the original PBLib. Compared to our own solutions, the flattening time and the metrics was significanlty lower in most cases for PBLib than for our own implementations of the same methods, which could be due to implementation differences and the bigger variety of available methods. One clear advantage to this approach is that we don't need to represent every single clause as a new `bool_clause` object and write those out to the CNFlatZinc; instead, new variables and clauses produced by PBLib are added directly to the internal model of the SAT interface.

For optimization, we require Max-SAT solvers. The WDIMACS format specified by the Max-SAT competition is similar to DIMACS, but (most importantly) precedes clauses with an positive integer for their weight and adds a `top` weight integer to the `p`-line. This `top` weight is the maximum cost of a solution, i.e. the sum of the unsatisfied clauses has to be strictly less than `top`. If we set `top` equal to the total sum of the weighted `bool_clause`'s plus one, then we can add our regular `bool_not` clauses as *hard* clauses by giving them weight `top`.

As for the weighted (unit) `bool_clause`'s produced by the objective, we need to negate the clause if the objective is to minimize. If a soft clause is already satisfied, we already add the weight to the objective value. If it is unsatisfiable, we can remove the clause.

4.3.2 SAT solution parser

The (Max-)SAT solver output specified by the SAT/Max-SAT competitions has a similar style as the input, here denoting the solution assignment $p \wedge \overline{q} \wedge \overline{r}$ of our input model in section 4.3.1:

```
c My SAT solver
s SATISFIABLE
v 1 -2 -3 0
```

The `c`-lines are comments, the `s`-line is the solver status and the `v`-line is the solution assignment. For Max-SAT solvers, the format is again similar but also includes an `o`-line which indicates the cost c (the sum of the weights of the unsatisfied clauses in the assignment). However, we want the objective value o (the sum of the weights of the satisfied soft clauses, the *profit* in some sense). For maximization, o is the sum of the weights of all soft clauses minus c. For minimization, $o = c$ since we inverted the soft clause variables, which flips the (unit) soft clauses from satisfied to unsatisfied and visa versa, which makes c actually the sum of the weights of the satisfied clauses.

The MiniZinc toolchain sends the solver output in chunks to the solver output handler, so the parser should be able to handle this efficiently and correctly. Even though the language of the solver output is quite simple, we require a parser with a good theoretical foundation, or a parser generator such as Bison (used for MiniZinc itself, but in this case overkill). We implemented a C++ version based on Rob Pike's talk titled *Lexical Scanning in Go*, which keeps

two *cursor* variables: `start` and `pos`. The `start` cursor lags behind the `pos` that is advanced until a full token is recognized (at a token delimiter, usually a line ending) and commit, which resets the `start` to `pos`. If the current chunk of data ends, everything before `start` (which has been committed) can be discarded, leaving the last bit to be picked up when the new chunk arrives.

For each parser state, we implement a state function that calls parser operations (`next`, `peek`, `commit`, etc.) that changes the `pos` and `start` variables, until at some designated point the state function will return a function pointer to the next state function. For instance, if we are in the middle of parsing a `s`-line, the solution assignment state function accepts new characters and assigns them to the internal SAT model, until it reaches end-of-line character `\n`, at which points it returns a pointer to the function that parses the next line indicator. If it it had encountered an end-of-file character `\0` instead, it returns a function pointer to itself, so it remains in the same state when the next chunk arrives. The parser can just keep calling the subsequent function that the current function returns a pointer to, which is very efficient.

Finally, after a solution has been parsed, the assignment is mapped back from DIMACS IDs to MiniZinc `VarDecl`'s which are then assigned to an output model which can then be printed to the user.

4.3.3 SAT solution output

The SAT solver interface has some extra control and features which can be accessed through the following command-line flags:

-a, --all-solutions When this flag is enabled for satisfaction problems, the found solution ($x_1 \wedge \overline{x_2} \wedge \cdots$) is (after outputting) negated, which yields a single *no-good* clause ($\overline{x_1} \vee x_2 \vee \cdots$), which is added to the model. This addition makes that particular solution illegal, and the SAT solving process is restarted. This process is repeated the solver returns UNSAT, upon which all solutions have apparently been found and we can output the *search complete* message. This is not very efficient, but very useful for unit tests which use small models. One way to improve this would be to only add output variables to the no-good.

--intermediate-solutions While normally this falls under the `--all-solutions` flag as well, we have separated it for now. According to the WDIMACS output format, solvers are not required to output intermediate solutions, and most only output intermediate costs (`o`-lines). This means this flag is only useful for obtaining the intermediate objective value to monitor progress, as the output solution just shows the minimum bound of all output variables.

--time-limit/--fzn-time-limit/--solver-time-limit These limit the maximum allotted time for compilation and solving combined, or (respectively) separately. Some solvers listen to certain interrupt signals, and (for Max-SAT) will return their best solution so far. But there is no guarantee that every solver

behaves this way (especially for solvers that compete in the complete track, were non-optimal solutions are not awarded any score).

--sat-cmd/--sat-flag/--sat-flags These determine how the backend solver will be called, and allows the user to pass extra flags to them to alter the solver's behaviour.

--solver-input-format Whether the input format of the solver is DIMACS or WDIMACS. This could be different from the problem type (satisfaction or optimization), as a Max-SAT solver could support satisfaction problems, or SAT solvers optimization problems if we had our own optimization strategy (e.g., branch-and-bound) that uses repeated solve calls. In the future, more formats could be added here (such as a PB format).

--output-dimacs-to-file Saves (W)DIMACS file to a permanent location (or to `/tmp` if flag is absent)

5 Evaluation (experiments and discussion)

In this section, we will show and discuss the experimental results in two parts. In section 5.1, we will isolate the channeling constraints and the (reified) set constraints into smaller benchmarks to demonstrate their individual merit and to make configuration decisions. In section 5.2, we will compare in detail the performance of each `mzn-sat` configuration and four control solvers using the MiniZinc Challenge 2019.

Re-using findings from experiments on an older[11] version of `mzn-sat`,we will use the (incomplete) Max-SAT solver `tt-open-wbo-inc` as backend, which won first place for the weighed categories on the incomplete track in the Max-SAT 2019 competition.

5.1 Evaluation of two `mzn-sat` features

Since these experiments are meant as a proof of concept and since we are mostly interested in the static metrics, they are small enough to be run locally on my MacBook Air with a *Intel(R) Core(TM) i7-5650U CPU @ 2.20GHz* CPU, and 16 GB of RAM, using a 5 minute timeout.

5.1.1 Channeling constraints

In section 4.1.4, six channeling constraints were introduced (two per encoding). Compiling a model that encodes an integer x with a domain size of $D(x) = [1, n]$ for two given encodings (and thus performs the appropriate channeling), we can get an idea of the static metrics and a rough indication of the flattening times.

[11] We made the decision to use the `tt-open-wbo-inc` solver for all `mzn-sat` configurations based on the results from an experiment that was run with an older version of `mzn-sat`, which due to time constraints we opted not to re-run for the latest version. Probably, the same conclusion applies that `tt-open-wbo-inc` is a good choice as backend solver, and that it does not seem at the moment that any particular solver benefits from a specific encoding more than any other solver. The details of this experiment can be found in appendix J.

Table 2: This table shows for different domain sizes the performance of each of the six channeling constraints, which form three pairs of competing methods. The *literals* columns shows the number of unique literals, the *clauses* show the number of clauses, *clause literals* the sum of the literals of all clauses, *time* shows the time to solve the model (which should be very close to the compilation time).

n	Channeling constraint	Method	Literals	Clauses	Clause literals	Flattening time (s)
100	direct ⇔ order	direct	231	651	1498	0.33
		order	233	555	1208	0.11
		pb	10332	30650	86294	0.22
	direct ⇔ binary	binary	152	994	2096	0.15
		elimination	152	994	2096	0.14
		pb	6026	17912	48179	0.13
	order ⇔ binary	subset	117	331	1169	0.10
		segment	117	330	859	0.10
		pb	5270	15588	46236	0.13
1000	direct ⇔ order	direct	2085	6181	14358	0.32
		order	2087	5185	11368	0.30
		pb	1103812	1655775	3863410	7.65
	direct ⇔ binary	binary	1115	12242	25498	0.50
		elimination	1115	12242	25498	0.57
		pb	884642	1327846	3098156	6.02
	order ⇔ binary	subset	1026	3049	14111	0.23
		segment	1026	3045	8089	0.23
		pb	115981	173484	404461	1.35
5000	direct ⇔ order	direct	10173	30405	70806	1.48
		order	10175	25409	55816	1.11
		pb	9941190	14911935	34794374	228.25
	direct ⇔ binary	binary	5212	75484	155988	13.63
		elimination	5212	75484	155988	15.52
		pb	8062094	12098464	28227732	57.28
	order ⇔ binary	subset	5035	15067	85149	1.04
		segment	5035	15064	40131	1.03
		pb	-	-	-	timeout

Based on these results, we conclude that the *general* PB methods are clearly outperformed by the encoding *specific* channeling constraints. Furthermore, any extra variables used by the non-PB channeling constraints are due to the unavoidable encoding's own constraints (e.g., the `exactly-one` for the direct encoding), since the those channeling constraints dot not use extra variables.

For the direct and order methods for the direct $\Leftrightarrow$ order channeling are very close and might still be equivalent. In any case, the second method is somewhat better in all metrics. The binary and elimination methods for the direct $\Leftrightarrow$ binary method seem to be equivalent, although the first implementation (binary method) has a slightly faster flattening time. It is notable that the flattening time and the number of clauses are *much* worse than for channeling constraints between other encodings. This should be investigated in future work. For order $\Leftrightarrow$ binary, the encoding specific methods seem to perform distinctly different, with the segment method being the winner. While the number of clauses are generated, but each contains only three literals. Mildly surprisingly, at the current experiment size, the flattening times are practically equal, even though the implementation of the subset method uses a new `binary` built-in to construct fixed binary strings, while the segment uses a more arithmetical approach without using that built-in, which we expected to be significantly faster.

Based on these results, the `mzn-sat` library will be configured to use the *order*, *binary* and *segment* methods to create channeling constraints for the other experiments.

5.1.2 Set and reified set constraints

As we have seen in section 4.2.3, for reified set constraints ($x \in S \Leftrightarrow r$) we either use equality constraints for x, or we convert S to an array of contiguous ranges, so that we can constraint x using inequality constraints on the bounds of the ranges. Fewer and larger ranges should benefit the order encoding over the direct encoding, and to some degree also the binary encoding. Element constraints convert their arrays to multiple sets (one per distinct) value.

The following (slightly convoluted) model creates a set with cardinality n (possibly $n+1$) which tends to larger ranges as $0.5 \geq p \geq 1$ increases. In other word, a range will be broken with a probability of $1-p$. We can't just omit set elements, since that would change the cardinality of the set, so we push them up by n instead. First we validate the non-reified set redefinition, which should remove variables (but not be influenced by p):

```
set of int: S = { if bernoulli(p) then i else i + n
  endif | i in 1..n } union {2*n};
var min(S)..max(S): x;
constraint encoded(x); % encodes x with the benchmark's
  encoding
constraint x in S; % uses set_in(x, S)
```

In the second test, we replace the set constraint of the last line with a reified set constraint for the new Boolean variable r.

```
var bool: r;
constraint x in S <-> r; % uses set_in_reif(x, S, r)
```

Table 3: This table shows sets of size $n = 100$ and different values for p the number of *literals*, the number of *clauses*, and the number of *clause literals* (the sum of the literals of all clauses). The first three rows are for the non-reified set constraint (for which the results were the same for any value of p), and the lower nine rows are for the reified set constraint.

Encoding	p	Literals	Clauses	Clause literals
direct		134	259	616
order	$*$	99	98	196
binary		21	141	894
direct	0.5	246	584	1465
	0.8	246	584	1465
	0.95	246	585	1468
order	0.5	248	393	883
	0.8	234	337	743
	0.95	206	225	463
binary	0.5	122	1042	2895
	0.8	136	949	2588
	0.95	71	243	600

If we look at Table 3, we can confirm that the number of direct and order encoding variables (from the original $|D(x)| = 200$) is reduced for the non-reified set constraint, since they are being fixed to false or unified, respectively. For the reified version, we see that p has no influence on the result for the direct encoding. As expected, as p approach 1.0, the order encoding becomes more efficient, but it is notable that even in the worst case where $p = 0.5$, the order encoding outperforms the direct encoding. This is simply because it does not require the `exactly-one` cardinality constraint, but only $|D(X)|$ clauses (of size 2) for its *ordered* constraint. A second iteration of this experiment should isolate the variables and clauses that are created due to the set constraint. The binary encoding does perform much worse for lower values of p, which means that either the (reified) lexicographical constraints need to be optimized if possible, and/or the very simple heuristic (inequality constraints for range sizes greater than 7) for choosing between equality and inequality constraints needs to be improved.

Based on these results, we have set the preferred encoding of set constraints to order. In future experiments, we would note separate the number of variables and clauses introduced by the set constraint from those introduced by the encoding of x. Furthermore, the `element` constraints that use the `sets` should be investigated as well. Finally, in the future we should try to calculate the expected value without the need for experiments.

5.2 Evaluation of `mzn-sat` and existing back-ends

In this comprehensive experiment, we test four test configurations of `mzn-sat` against four control solvers. Each configuration is only different in the *encoding mode*. The `mzn-sat-direct`, `mzn-sat-order` and `mzn-sat-binary` encoding modes use exclusively direct, order and binary encodings, respectively. The `mzn-sat-mixed` configuration will use the constraint's preferred encoding according which are set to the constraint's category's default setting as listed in section 4.2.1, if variables do not have an encoding (which then might lead to different encodings in the same model, which might lead to channeling constraints). If an encoding does already exist, it will re-use it, if possible.

As control solvers, we chose `chuffed` (LCG) (version 0.10.4) and `picat-sat` (SAT) (version 2.8#6) solvers as they use SAT solver technology as well. We added `gurobi` [34] (version 9.1.0), a high-end MIP solver and Gecode (development branch, ref `431520083a51fc2f31c22fbc7b0378e7a1588e42`), an open-source CP solver, for some added context of the viability of SAT in general for the problem. The four `mzn-sat` configurations and the four control solvers were run on the full MiniZinc Challenge 2019.

The experiments were run with a 20 minute timeout on a single-core *Intel Xeon 8260* CPU (non-hyperthreaded) with 4 GB of RAM made available, on a compute server courtesy of Monash University.

We examine the results on three different levels. In section 5.2.1, we aggregate for each solver all the instances of all models in Table 4. In section 5.2.2, we aggregate for each model all the instances in Table 8 in appendix K. In 5.2.3, we plot the solver's reported objective value over time for some specific instances.

5.2.1 Per-solver evaluation

On the `inc` and `err` statuses First, to explain the non-zero results in the `inc` column. If the solver reports a `sat` or `opt` status, then we check the final solution with a checker solver (`gecode` on the MiniZinc release branch, version 2.5.4) by solving the model with the decision variables fixed to their values in the given solution, where a `sat` or `uns` status from the checker tells us that the solution is a correct or incorrect solution, respectively, to the instance (according to the checker solver). It does not check whether the reported objective value is consistent with the solution assignment. The checker reported correct results for all instances and all solvers.

Next, we check for contradictions in two ways. If a solver reports a `uns` status, while another solver proved the instance feasible by outputting a correct solution, then the `uns` status counts as incorrect. This is the case for 5 of the 7 `uns` cases for `gurobi`. What is causing these false `uns` statuses is currently unknown. It might be a configuration problem or it is possibly caused by changes on the `mzn-sat` development branch.

Additionally, we check that the objective value of every optimal solution doesn't contradict objective values of other (optimal or feasible) solutions. For one instance of the minimization problem `stack-cuttingstock`, `mzn-sat-order`

Table 4: Aggregate results from four test (`mzn-sat` with different encoding modes) and four control solvers solving the MiniZinc Challenge 2019, which counts 20 models, each with five instances. The first column shows the configuration tested. The next four columns show the breakdown of statuses output for the instances: the `opt` status indicates that optimality was proven on the best found solution, `sat` that the instance was feasible but without optimality proven, `uns` that instance is unsatisfiable, `unk` for timeout reached before a feasible solution was found and `err` that there was an error of some kind. Then, the `inc` column shows how many of the `opt`, `sat` and `uns` statuses were incorrect, and the `fail` column the amount of instances where the solver failed to produce a correct solution (the sum of the previous three columns).

configuration	opt	sat	uns	unk	err	inc	fail
mzn-sat-mixed	14	21	0	12	53	0	65
mzn-sat-direct	13	18	0	13	56	0	69
mzn-sat-order	22	20	0	9	49	1	59
mzn-sat-binary	34	24	0	9	33	0	42
gecode	29	49	0	20	2	0	22
gurobi	48	28	7	7	10	5	22
chuffed	56	35	1	6	2	0	8
picat-sat	40	26	2	25	7	0	32

outputted a (correct) solution which it reported as optimal but which had an objective value of 17. However, other optimal solutions were reported the control solvers with an objective value of 16. We have more or less pinpointed the bug, but it couldn't be fixed due to time constraints. This does leave open the possibility that the intermediate objective values reported by `mzn-sat` (see section refsec:eval-per-instance) could be off, but since all other optimal solutions of `mzn-sat` (and other solvers) are consistent this possibility is remote.

All `err` statuses for both test and control solvers are caused by out-of-memory errors, except for three cases in which the `mzn-sat` process somehow ran out of disk space. The former problem is a systematic one, while the latter can be safely ignored for the evaluation. The available disk spaqce is shared as the benchmarking jobs run in parallel, so most likely there was some unfortunate peak of combined writing between clearing.

Evaluation of the encoding modes The encoding mode makes a significant impact on the performance of `mzn-sat`. The binary encoding is the clear winner, both in terms of consistency (it finds a solution in over half the instances) and the number of optimal solutions. The next best is the order encoding, for which we observe a sharp increase of `err` statuses as more memory is required to encode the instance, while the number of timeouts is unchanged. For the direct encoding, the number of out-of-memory errors grows by seven, and the number of timeouts by four.

Our speculation is that the relatively heavy auxiliary and clause require-

ments for the exactly-one constraints for the direct encoding is the most likely suspect, but a memory profiling tool is needed to confirm this. The constraint might also be slow to encode, causing more timeouts to occur in PBLib's executing after the flattening phase (but before the solver phase). This would fit with the fact that the number of timeouts did not increase for mzn-sat-order, as its constraint to keep the order encoding ordered is linear in time. For more context, the handling of timeout or memory error should be improved. There was never a need for any of MiniZinc's interfaces to catch timeouts before the solving phase, since this is a relatively quick process for other interfaces. It was out-of-scope to implement this global MiniZinc feature in this project.

The results of the mixed encoding mode lies in-between the order and direct encoding. Theoretically the mzn-sat-mixed encoding mode might have an advantage over the other modes in two ways. First, it might decide to use one encoding over the other because the constraints in the model favours it. If it chose correctly (and only uses that encoding), the results should be as good as the best of the other encoding modes. The other advantage is that a SAT model in which multiple encodings are mixed might have a propagation advantage that outweighs the channeling constraint overhead.

From these results, it is hard to say if the mixed encoding mode is better than the direct encoding because the presence of multiple encodings is beneficial (despite the channeling constraints), or if for some models the constraints picked the binary or order encoding and re-used them throughout the model. This should be tested on a smaller scale first, once mzn-sat-mixed offers better control over and introspection of the encoding choices.

Evaluation of the test and control solvers Compared to the control solvers, we first have to note that due to presence of two inc statuses (due to contradicting optimal solutions), the solutions of mzn-sat should still be checked (but as we see with gurobi which shows 7 incorrect uns statuses, this is a good idea for any solver). Furthermore, mzn-sat at this point is not a very consistent option compared to the available control solvers. Of course, picat-sat, the backend most like mzn-sat(-binary), is the most inconsistent control solver with 32 failed instances.

Comparing mzn-sat and picat-sat, we see that the big difference is that the fail count for picat-sat mainly consists of unk statuses, while for mzn-sat it mainly consists of (out-of-memory) err statuses. It seems that generally, picat-sat runs out of time whereas mzn-sat runs out of memory. In some sense, this is the better problem to have as memory issues can be tracked down straightforwardly through profiling or introspection to find out which constraints take up the most space, while improving the solving time of the SAT model is less straightforward. The question is, how many of err statuses of mzn-sat can be converted, and will they become unk or sat/opt? Both mzn-sat-binary and picat-sat marginally outperform a pure CP solver like gecode, which shows that SAT is still a good choice over CP for specific models (just not most).

The LCG solvers chuffed performs extremely well, even better than the commercial class MIP solver gurobi. In a separate experiment not shown here, mzn-sat was run with an additional pre-optimization feature of MiniZinc. If enabled, MiniZinc will (in addition to its regular pre-optimization) execute an additional pass over the model with gecode, which reduces variable domains. This combines for mzn-sat the CP propagation and SAT solving somewhat in line with chuffed. However, this made result worse for mzn-sat. More granular experiments are needed, so it is too soon to give up hope that some pre-optimization might help mzn-sat, as it seems a good fit.

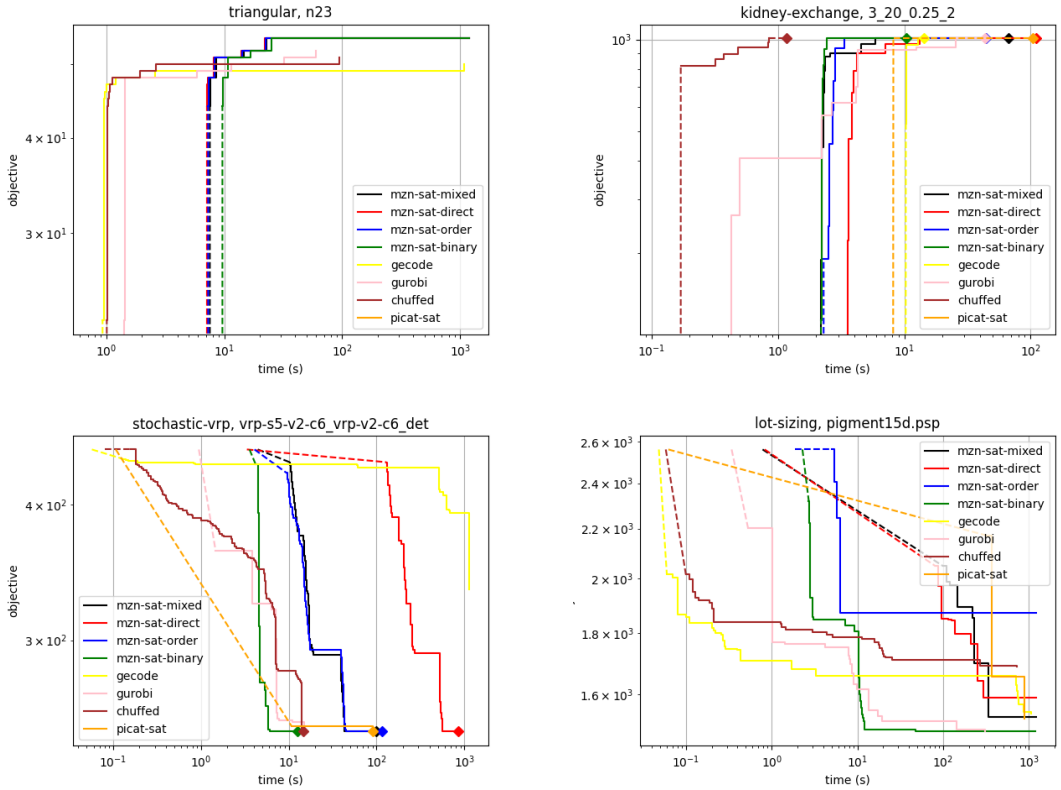
5.2.2 Per-model evaluation

In Table 8 in appendix K, we can pick out some models where mzn-sat performs compared to one or more control solvers. It is hard to discover general patterns without intimate knowledge of the models, so we can't make specific claims about which type of model fits which type of solver. For this, improved tooling is required.

In accap, we see that mzn-sat-binary finds 2 opt, 1 sat and 2 err statuses, while gecode has sat for all instances. This tells us that for smaller instances, SAT has an advantage over CP when it comes to finding the best solution, but for larger instances CP is more reliable. The results for SAT seem to be more spread out, while CP is more consistently sat. We see the same for fox-geese-corn, hrc, ptv and stack-cuttingstock. We see picat-sat and mzn-sat performing badly on code-generator, groupsplitter, liner-sf-repositioning, nside and rcpsp-wet-diverse. This suggests that these models are perhaps less suitable for pure SAT encoding.

5.2.3 Per-instance evaluation

In this section, we showcase a (hand-picked) selection of interesting runs of the test and control solvers by plotting the solver's reported objective value over time. In these graphs, mzn-sat performs generally well or shows some other noteworthy behaviour. We don't show instances in this section where mzn-sat performed poorly, because in those cases there usually was no solution by mzn-sat so the graphs are not interesting.



The graphs in this section show for every solver the time on the x-axis and objective value on the y-axis of every given intermediate solution. Both axes are in logarithmic scale. If there is no line for a given solver, then it did not find any solutions. The start of the line indicates when the solver has finished flattening[12], followed by a dashed line to the first solution. Since there is no absolute reference for the objective value before there is a solution – and the lower bound for the objective would be a poor choice since it usually is much lower than the actual solution – the height of the flattening time is at the same height as the worst solution (of all solvers).

`triangular` Out of all the MiniZinc Challenge 2019 problem, `triangular` is by far the simplest model, but proving optimality becomes intractable as its single parameter n grows. All variables are 01-integers with linear constraints, so the model itself is essentially a Pseudo-Boolean model. The `bool2int` function

[12]The flattening times of `mzn-sat` do not include the possibly substantial encoding time of `PBLib`, which takes place after flattening. In the future, a *compilation* time should be recorded when the CNF model has been written out. Of course, `picat-sat` has the same behaviour, since it uses the `std` library for flattening, and the SAT encoding comes after.

(if properly implemented) should represent a 01-integer as a single Boolean variable regardless of the encoding setting, so the behaviour of each `mzn-sat` configuration is nearly identical. Furthermore, the linear constraints are actually at-least-one Boolean cardinality constraints, which can be represented as a single clause each, making the model very suitable for a MaxSAT encoding.

Even though in this case the encoding is simple, a purely SAT based technology needs to finish the encoding process before the solver can find a first solution. For `picat-sat`, the flattening process never finishes for this instance. So, if one wants a good solution quickly, other technologies are a safer bet unless the model is actually Boolean (and not just Pseudo-Boolean). However, once the encoding is completed, `mzn-sat` quickly catches up and in the end finds the best solution out of all solvers. In the other instances except the two largest ($n = 29$ and $n = 37$), `mzn-sat` is best or shared best.

`kidney-exchange` In this fairly typical instance, we can see that all encodings are viable in the order we observed in the per-solver evaluation in section 5.2.1. The binary encoding performs best, followed by order and the direct encoding comes in last. The mixed encoding is usually very similar to one encoding, usually other. Alternatively, as in this case, it nestles between the direct and order encoding. For now, we should assume that the mix consists of direct and order encoding, in which every direct encoding variable would have been better off being an order encoding variable. Out of all the control solvers, only `chuffed` solver performs substantially better than the best `mzn-sat` configuration.

`stochastic-vrp` For this vehicle routing problem, we see the same typical behaviour between encodings as in `kidney-exchange`. This time, `mzn-sat-binary` does proof optimality faster than all control solvers by a narrow margin in all but one instance. We see a tied first place with `chuffed` and `gurobi`. In other instances for this model, `mzn-sat-binary` and `gurobi` trade first place, while `chuffed` is sometimes much better, sometimes much worse. The mixed encoding was very conservative, as its almost identical trajectory to `mzn-sat-order` suggests a purely order encoding. The binary encoding would have been a better choice, but at least it didn't opt for a direct encoding.

`lot-sizing` This job-scheduling type problem is a lot more complex, and includes global constraints such as `global_cardinality`, `at_least`, `at_most` and `all_different`. All of these use new redefinitions in the `sat` library. The binary encoding does well, but this model is unique in that the direct encoding outperforms the order encoding. Usually, this is the other way around. It should be investigated whether this is a fluke in the order encoding, or if this model is indeed better suited for a direct encoding. The mixed encoding also shows some new behaviour that cannot be explained without better introspection.

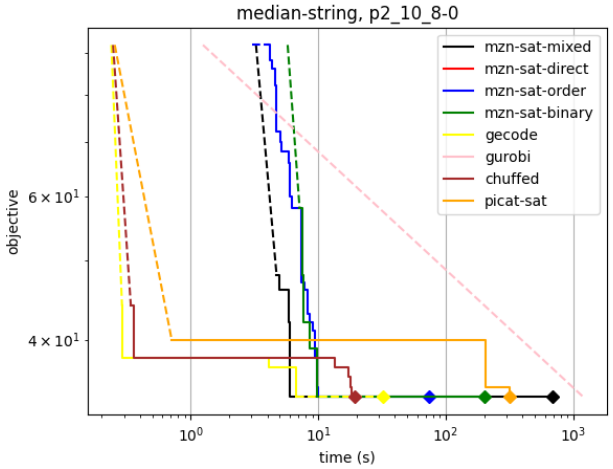


Figure 2: The per-instance benchmark results using an older version of `mzn-sat`

`median-string` In the penultimate version of the `mzn-sat` solver, we observed an anomaly in the median string problem. The latest version of `mzn-sat` (used in the evaluation up to this point) no longer shows this behaviour.

All`mzn-sat` except `mzn-sat-direct` are somewhat viable, finding the best solutions quickly, although they finish proving optimality later than most control solvers. The `mzn-sat-binary` manages to finish sooner than the binary encoding implementation of `picat-sat`, and we find that an order encoding for this model works a little better than the binary encoding, which is rare but not unique. While the variables for the `median-string` model are not 01-integers, the domains tend to be small, and the constraints are mostly inequality and maximum constraints, with a couple of equality constraints, which suits the order encoding well.

The big revelation is that the `mixed` encoding actually finds the best solution the quickest out of all solvers, although we cannot be sure for `gurobi` as we are missing its intermediate solutions due to a now solved problem. The `mzn-sat-mixed` finds the best solution a few seconds before `mzn-sat-order` does. Anti-climatically, it then spends the remaining 99% of its time proving optimality, finishing just a couple of minutes before the timeout.

In some cases (like `triangular`), due to its tendency to re-use encodings, we expect `mzn-sat-mixed` to generally choose and stick to one encoding. For `triangular`, this would be the order encoding because the first constraint that is flattened is an inequality constraint, so `mzn-sat-mixed` will produce the same SAT model as `mzn-sat-order`, with the same performance. For this one model, however, it displays different (and in one aspect, superior) behaviour. A look at the debug variable annotations – `expression_name(...)`, our

only rudimentary introspection at the moment – of the flattened model (see Listing 2) of `median-string` confirms that a mixed encoding of direct and order encoding was indeed used, to some effect.

Listing 2: The flattened model of `median-string` for instance `p2_10_8-0` which shows a mix of direct and order encodings through `expression_name` annotations

```
[..]
var bool: X_INTRODUCED_150_ ::var_is_introduced ::
  expression_name("X_INTRODUCED_142_>=2");
var bool: X_INTRODUCED_151_ ::var_is_introduced ::
  expression_name("X_INTRODUCED_142_>=3");
var bool: X_INTRODUCED_169_ ::var_is_introduced ::
  expression_name("x==0");
var bool: X_INTRODUCED_170_ ::var_is_introduced ::
  expression_name("x==1");
[..]
```

The flattening time of `mzn-sat-mixed` is equal to the `mzn-sat-order` (and only slightly faster than `mzn-sat-binary`). We expect a (future) breakdown to show one of two explanations: either most (or the larger) integer variables are order encoded, since otherwise we would have seen a longer flattening flattening time due to the `exactly-one` of the direct encoding variables and the addition of the channeling constraint. Alternatively, this overhead has been effectively offset by the faster encoding time of the direct encoding constraints that have replaced the order encoding constraints.

6 Conclusion

In this thesis project, the aim is to design and implement a CP-to-SAT interface, using the unique capabilities of the MiniZinc modeling language. The experimental results show the promise of, but also the long road ahead to, a *truly* universal SAT interface. We can see that for a good amount of models, everything falls into place and we are able to achieve very good results, without having had these specific models in mind from the start. Furthermore, some novel techniques have proven some merit about which we can be carefully optimistic. In other models, either simple or complex, a particular constraint might trip up the whole process to create a slow or wasteful encoding. More engineering and/or techniques are required to reduce the amount of models where this happens (as other SAT solvers are still more reliable), but we should be aware that a purely SAT based solver interface remains somewhat of a glass canon.

If one wanted to use `mzn-sat` today, the main learnings from the evaluation are that due to the encoding process, a first solution is usually late, but good solutions arrive quickly after, sometimes sooner than other solvers. The final race to proof optimality can go either way. For the largest instances, `mzn-sat` is not a good solution as it will run easily out of memory. The binary encoding is the safest bet, but for some rare instances the order encoding might work better. The `mzn-sat-mixed` configuration lacks introspection and control to leverage any mixed encodings well. On top of that, its current tendency to re-use (rather than re-encode) means that the instances in which it performed decently might be simply equivalent to what e.g. `mzn-sat-order` would produce. However, one lucky exception has shown that the idea does have potential, which is exciting.

In conclusion, some big steps have been made. The interface is functional, compatible with SAT and MaxSAT solvers and offers the user some options, and outputs progress, statistics and visualizations. The library has been extensively tested for gives correct results, barring one instance. Many algorithms have been implemented that brought the library to the next step, and some interesting ideas for encoding particular constraints are implemented and might be improved in the future. A good amount of quite difficult instances run, with some on par or exceeding existing solutions. In addition, the dynamic encoding system, while lacking in depth, provides MiniZinc users to experiment and solve CSPs. Improving the breadth and introspection of this system is interesting future work.

7 Future Work

The following topics can be considered for future work, in addition to the specific improvements mentioned in the footnotes throughout this report:

- Performance:
  - Some run-time or compiler bugs remain that need to be fixed so at least the MiniZinc Challenge 2019 does not have trivial errors or incorrect results.
  - The amount of failed (especially due to out-of-memory errors) should be reduced through profiling.
  - More benchmarks should be added for specific, important constraints, to further analyze and improve them.
- Functionality:
  - A basic implementation of an optimization strategy for SAT solvers (such as branch-and-bound), so that the optimization problem can conveniently be solved with SAT solvers
  - Integration with the SAT heritage project [32], so that any SAT solver is accessible
  - Better static analysis of the SAT model
  - Duplicate clauses could be detected, as they might indicate sources of redundancy
- Dynamic encodings:
  - Fine-grained control over the choice of encoding per constraint and/or per variable. Beside the initial choice of variables, the choice between re-use and re-encoding should also be adjustable.
  - A cut-off limit of the domain size could be added to choose between binary or other encodings.
  - Reporting on which constraint encoded which variable and how. Since there our implementation has an dependency on the flattening order, the user can influence this to some extent if there were introspection tools available.
  - One-way encodings should be separately evaluated.
- Other SAT techniques:
  - The two's complement addition and multiplication properties as well as those for direct and order encoding could be used to simplify linear constraints and solve arithmetical constraints (see appendix G).
  - The pre-emptive partial encoding should be evaluated

Appendix

In this appendix, we discuss some ideas that didn't make it into the version of the project, or were implemented but replaced by others, or were implemented but not quite ready to be evaluated, but might still be worthwhile to look at in the future.

A A less redundant binary encoding scheme for signed integers

The following alternative binary encoding function f for x binary encoded with $n = \lceil \log (D_u(x) - D_l(x) + 1) \rceil$ Boolean variables $b_0, b_1, \ldots, b_n$ for $0 \leq i \leq n$ is defined as that $f(x, i)$ returns b_i, the i'th most significant bit of the bit string that encodes $x = D_l(x) + \sum_{i=0}^{m} f(x, i)2^i$.

This encoding is similar to the order encoding in that the lowest index denotes the *offset* of the encoding which is added to the sum of the value of the true variables. With order encoding, each true variable contributes a constant value of 1, while with binary encoding, the bit's position (not the index!) *bit* determines the value. This way, there is no need for a sign bit, as negative for integer values the offset can be made negative. There is also less redundancy, as the encoding range is $[D_l(x), D_l(x) + 2^n]$, which means the lower bound always lines up exactly with the lowest value, $000\ldots$. Domain gaps and the upper bound (if less than 2^n) will still need to be constrained.

This encoding should work as well as the two's complement scheme for expanding integer linear expressions into PB constraints in section 4.1.3, and channeling constraints can likely be adjusted for this as well. However, the problem starts when constraining two binary with different offsets, as the properties in section 4.1.1 no longer hold: the binary encoding are out of sync with each other and two identical assignments no longer represent the same integer value. If the offset is a power of 2 apart, then this can be fixed by a bit shift, but otherwise it becomes complicated.

B One-way channeling constraints

All channeling constraints shown in this project are two-way, which means that information flows bi-directionally between the encodings. When a two-way channeling constraint is in place, the each value in the domain will correspond to exactly one assignment of one encoding, and one of the other encoding, since our encodings do not have redundant assignments that map to the same value. Simply put, the constraints will make sure that the assignment of x and of either encodings looks like Table 1. This is in most cases beneficial for propagation, but it is not necessary for correctness of any model.

A correct alternative would be to use *one-way* channeling constraints, which are generally lighter on clauses and variables. Many of the two-way channeling

constraints can be converted to one-way constraints by omitting an implication direction. If there are 2 constraints on x where the first uses the order encoding and the second the direct encoding, and we create a one-way channeling constraint from the direct encoding to the order encoding, then both constraints will be enforced on the order encoding, while the direct encoding still only reflects the second constraint and thus might have assignments that are not consistent with the first constraint.

Yet, this inconsistency in the direct encoding poses no problem as long as we map the value of x from the assignment from the order encoding, and not from the direct encoding. All legal assignments of x and its encodings are shown in Table 6, and it's clear that at least the order encoding always corresponds to the assignment of x. In general, the channeling constraints between any number of encodings of x form the edges of an implication graph. As long as there is a path (of one or two-way channeling constraints) from every encoding to one *root* encoding, then that root encoding will always be safe to read the assignment of x from. In addition, the assignments of the direct encoding are at most $\mathcal{A}(x)$, so it is partly correct.

The downside of using one-way channeling constraints would be that the first constraint does not receive information from second constraint, but it depends on the model whether this harms propagation (although we suspect that it generally will).

Table 6: All assignments of x with domain $D(x) = [2, 4]$, which has two encodings, direct and order, that are channeled through a one-way channeling constraint from the order encoding to the direct encoding. Note that in this case, the order encoding is always correct, but the direct encoding might not be (but does obey that $\mathcal{A}(B_{\text{direct}}) \leq \mathcal{A}(B_{\text{order}})$).

$\mathcal{A}(x)$	$\mathcal{A}(B_{\text{order}})$	$\mathcal{A}(B_{\text{direct}})$
2	100	100
3	110	100
3	110	010
4	111	100
4	111	010
4	111	001

C The `bimander` encoding for the at-most-one cardinality constraints

An at-most-one Boolean cardinality constraint fixes a finite set X of of n Boolean variables to have at most one true element of X is true. In section 2.3 we have already seen that the Sudoku model can be modelled entirely with Boolean cardinality constraints using at-most-one constraints (and that at-least-one and thus exactly-one constraints are trivial). Since every direct encoding of an inte-

ger variable requires an at-most-one constraint with n equal to its domain size, it is important to have an efficient encoding. We handle the general case for integers in section E.

As mentioned, we can use the pairwise encoding:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg(x_i \wedge x_j) \tag{39}$$

While this creates no additional auxiliary variables, it requires $\binom{n}{2}$ clauses. From the literature, the best encoding we could find was the `bimander` encoding [35] which for parameter $m = \frac{m}{2}$ creates $n \log n - \frac{n}{2}$ clauses, and $\lceil \log n \rceil - 1$ auxiliary variables[13]. The `bimander` encoding is based on two existing methods, `binary` and `commander` encoding [36]. The idea is to partition X into a set of m groups G and constrain each group in G to have at-most-one one true variable. For this we use pair-wise encoding as the groups should be of manageable size depending on the setting of m. However, any at-most-one encoding can be used, including `bimander`.

Now, all that is left is to make sure that at most one of the *groups* has a true variable (as we don't have to worry about any individual group having 2 or more and violating the constraint). To this end, we create auxiliary variables $B = b_1, b_2, \ldots, b_{\lceil \log m \rceil}$. Then, for each group G with index $1 \leq g \leq m$, we create an implication for each variable $x_i \in G$ to each $b_j \in B$. However, if the i'th bit of the binary representation of g is 0, we post $x_i \Rightarrow \overline{b_j}$ instead of $x_i \Rightarrow b_j$.

Now, no two groups will have a true element simultaneously, since the x_i in each group will imply their own unique combination of b's and $\overline{b}$. If two (or more) groups have a true x, the auxiliary variables will contradict each other. For example, if one of the elements of group 3 is true, then because $3 = 10$ in binary, $b_1 \wedge \overline{b_2}$ is implied. If another group 4 also has a true variable, $b_1 \wedge b_2$ is implied which leads to the contradiction $b_2 \wedge \overline{b_2}$. This always happens, because each group links to a unique combination of all m auxiliary variables.

The algorithm has been implemented from scratch for this project, but I took some hints from the implementation in the `PBLib` library [33]. See appendix D for an idea for a new `mimander` encoding, an extension of `bimander` encoding for multiple overlapping at-most-one cardinality that memoizes the groups.

D The `mimander` encoding for multiple overlapping at-most-one cardinality constraints

Another extension of the `bimander` encoding was explored with a new `mimander` encoding, where our idea was that 2 (or more) at-most-one constraints on different but partially overlapping inputs X_1, X_2 can be combined. If the groups

[13] Another m value tested by [35] is $m = \sqrt{n}$, but we did not experiment with this or other settings

are cached between at-most-one constraints, then a group from X_1 (and its auxiliary variables) should be reusable for the at-most-one constraint on X_2 if it contains common variables $x \in X_1 \cap X_2$. A pre-condition for this is that the auxiliary variables for the existing have some range left in their binary encoding scheme (say if two b's are only encoding 3 groups in X_1). Furthermore, its benefit depends on the occurrence of overlapping at-most-one constraints in the model, and the likeliness that X_1 and X_2 are partitioned in a favourable way (a partitioning heuristic might help with this), which must all be weighed against the overhead of storing and searching the groups. Finally, because of the many ways two sets can overlap, we never got to a fully working encoding for this.

E Sorting networks for general Boolean cardinality constraints

A general Boolean cardinality constraint $|X| \# c$ fixes a finite set X of n variables $x_1, x_2, .., x_n$ to have at most, at least or exactly $c \leq n$ true elements, where c is an integer variable and $\#$ is the $\leq$, $\geq$ or $=$ comparator, respectively. We can use general Boolean cardinality constraints to create integer cardinality constraints which we'll discuss in section 4.2.5.

Sorting and merge networks
A sorting network takes as input n Boolean variables X and produces a (decreasingly) *ordered* output of n Boolean variables $S = (s_1, s_2, \ldots, s_m)$. For an array of Booleans, ordered just means the true literals are on the left. For instance, if the assignment of X contains two true literals and one false, S will be $s_1 \wedge s_2 \wedge \overline{s_3}$ no matter which of the three possible orders ($\overline{x_1} \wedge x_2 \wedge x_3$, $x_1 \wedge \overline{x_2} \wedge x_3$, $x_1 \wedge x_2 \wedge \overline{x_3}$) the assignment of X is in.

We have based the construction of the sorting network encoding on [37], in which no less than four approaches are introduced and combined into a new mixed approach, which is the main contribution of the work[14]. In this section, the definitions and algorithms are paraphrased (refer to the original paper all the details) to the best of our abilities and insofar relevant, in order to explain and motivate the changes.

- **Merge networks, recursive method** A merge network takes as input two (decreasingly) ordered sets X and X' of sizes a and b and returns a (decreasingly) ordered set S of size $n = a + b$. The recursive method constructs a merge network recursively through a merge-sort like procedure.

- **Sorting networks, recursive method** Using merge networks, a sorting network can be build recursively by splitting the input (of size n) at some position l, recursively sorting each part, and merging the resulting

[14] The algorithms have been implemented in `PBLib` [33], but have been re-implemented from scratch for this project.

sorted output with a merge network. This produces a sorting network of size n. We use $l = \frac{n}{2}$.

- **2-comparators** A 2-comparator can be seen as a tiny merge network (s_1, s_2) with $a = b = 1$, or a tiny sorting network of $n = 2$. It has two input variables x_1 and x_2 and outputs two new variables $s_1 \Leftrightarrow x_1 \vee x_2$ and $s_2 \Leftrightarrow x_1 \wedge x_2$, so that $s_1 \geq s_2$. Another way to think about 2-comparator is that they pass through their inputs, unless the inputs are in the wrong order ($x_1 \wedge \overline{x_2}$), in which case it performs a swap. In a sense, sorting and merge networks don't do any sorting, they just recursively split up their input until it is small enough ($n \leq 2$) to use a 2-comparator in the base case.

  A 2-comparators can be constrained with 6 clauses and 2 variables. Only the first 3 clauses (on the top row) are required for maximum cardinality, only the latter 3 for minimum cardinality, and all 6 for exact cardinality. Let's call these maximum, minimum and exact 2-comparators:

$$\begin{array}{lll} x_1 \Rightarrow s_1 & x_2 \Rightarrow s_1 & x_1 \wedge x_2 \Rightarrow s_2 \\ \overline{x_1} \Rightarrow \overline{s_2} & \overline{x_2} \Rightarrow \overline{s_2} & \overline{x_1} \wedge \overline{x_2} \Rightarrow \overline{s_1} \end{array} \tag{40}$$

Cardinality networks, recursive method

In the case that we are interested in the minimum cardinality of c, we will only care about the first c elements of the output of the merge or sorting network (and in the case of maximum or exact cardinality the first $c+1$). This can save a lot of variables and clauses, as often $c \ll n$. The paper introduces simplified merge networks and cardinality (sorting) networks which limit their output to c instead of n elements. Such networks we can collectively call c-cardinality sorting/merge networks, of which regular networks are a generalization where $c = n$ or $c = a + b$ respectively.

In our implementation, there is no separate cardinality network procedure, the regular network procedure is extended with a parameter c for its cardinality. Also, in the paper there are different cases 2 or 3 cases per network construction function (which separate odd/even input pairs for more convenient proofs, presumably) which can all be combined into one function.

From this point forward, the paper only considers maximum cardinality, not minimum or exact cardinality (and simply states that *similar constructions for the other constraints can be devised*). Luckily, for the most part this can be fixed by extending the definition of a 2-comparator to a c-cardinality 2-comparator as well (wherein $c \leq n \leq 2$). When we put these new c-cardinality 2-comparators in the correct spots in the algorithm, one can see that for the maximum cardinality case the algorithm is unaltered (because in a sense the original algorithm hard-codes the behavior of a maximum 1-cardinality 2-comparator), yet everything will now also work for minimum (and by extension, for exact cardinality).

- **c-cardinality 2-comparators** A 2-cardinality 2-comparator is a regular 2-comparator, but a 1-cardinality 2-comparator will only return one

output s_1 instead of two. So, only the clauses in formula 40 that have s_1 as consequent are needed, which are the first, second and sixth. This comes on top of the rule that for minimum or maximum comparators, we don't require all clauses either: so a minimum 1-cardinality 2-comparator should only create the first and second clause, and a minimum 1-cardinality 2-comparator only the sixth.

- **c-cardinality sorting networks, recursive method** The definition is as in the paper, but the base case 2-comparator now also receives c. Specifically, this means that for $n = 2, c = 1$, the sorting network return our new 1-cardinality 2-comparator.

- **c-cardinality merge networks, recursive method** The definition is as in the paper, but all 2-comparators now also receive c. Specifically, the base case 2-comparator might become a 1-cardinality if $a + b = 2, c = 1$, and if $a, b \leq c, a + b > c$ and c is even, the last output variable should be the output of a 1-cardinality 2-comparator.

The output size of a c-cardinality network or comparator is always the minimum of c and of the size of its input(s). A recursive method to calculate the amount of auxiliary variables and clauses the network will produce is presented in the paper, but this will be underestimating the amount of clauses since that number now depends on the comparison type of the cardinality constraint.[15]

Sorting, merge and cardinality networks, direct method
In addition to the recursive method, there is also a direct method to produce sorting and merge networks. This method can be extended to produce c-cardinality sorting/merge networks. The direct method needs fewer auxiliary variables, but at the cost of needing many more clauses. For small inputs, the direct method outperforms the recursive method. Again, we need to make some adjustments to let the c-cardinality version work with minimum and exact cardinality.

For these adjustments, it's important to see in general that implications like $x \Rightarrow s$ from an input variable x to an output variable s allow us to constrain the maximum cardinality of the input by posting the unit clause $\overline{s}$, while the implication $\overline{x} \Rightarrow \overline{s}$ allows us to constrain its minimum cardinality by posting s. This is somewhat counter-intuitive, as $x \Rightarrow s$ actually enforces that s adheres to the minimum cardinality of x, in order that $\overline{s}$ may limit the maximum cardinality of x.

- **sorting networks, direct method** This method iterates through the powerset $\mathcal{P}$ (the set of all subsets) of X. Then, for each subset Z_i with

[15] Unfortunately, I never got around to fixing this calculation (and the one for direct networks), which means our current sorting network will be making some sub-optimal choices later on in the mixed method (although the result will still be correct).

size[16] $k = |Z_i|$, we create an implication that if all variables in Z_i are true, then $s_k \in S$ must be true also.

$$\bigwedge_{Z_i \in \mathcal{P}(X) \setminus \{\emptyset\}} \left(\bigwedge Z_i \rightarrow s_k \right) \tag{41}$$

True to its name, this directly encodes that if, say, X has 3 true variables out of a total of 5, then one of its subsets Z_i consists of those 3 true variables which will imply s_3 to be true. Additionally, there must also be 2 smaller $Z_j \subset Z_i$ of sizes 2 and 3 subsets of size 1 which will then imply s_1 and s_2 to be true. Thus, S will correspond to the minimum cardinality of X, which means we can constrain the *maximum* cardinality c of X by adding $\overline{s_{c+1}}$.

However, we can't use the same network for minimum cardinality, since setting (for instance) s_3 to true – in the hope of constraining at least 3 true variables – will not change anything about X! Instead, we need to create implications that if the subset Z_i consists of k variables that are all *false*, then s_{n-k+1} is false (the +1 is easy to see by thinking about at most 0, in which case the Z_i with $k = n$ being all false must set $\overline{s_1}$). For exact cardinality, both the following and the previous constraint are needed:

$$\bigwedge_{Z_i \in \mathcal{P}(X) \setminus \{\emptyset\}} \left(\bigwedge_{z_j \in Z_i} \overline{z_j} \rightarrow \overline{s_{n-k+1}} \right) \tag{42}$$

- **c-cardinality sorting networks, direct method** For the maximum cardinality c of X, we only need a $(c+1)$-cardinality sorting network S of X, as $\overline{s_{c+1}}$ will tell us at most c are true. So, we can simply skip any subsets of size $k > c+1$. This also works for the minimum cardinality constraints c of x, where we can skip subsets of size $n-k+1 > c$. Simply put, for a c-cardinality network we should only ever add implications with consequent s_i's that we are interested in, which will be $i \leq c$.

- **merge networks, direct method** For a merge network with two ordered inputs (X of size a and X' of size b), we create $n = a + b$ auxiliary variables $Y = (y_1, y_2, \ldots, y_{a+b})$ which will correspond to the minimum cardinality of X, so that we can limit the maximum cardinality. Then we add implications:

$$\{x_i \Rightarrow y_i, x'_j \Rightarrow y_j, x_i \wedge x'_j \Rightarrow y_{i+j} : 1 \leq i \leq a, i \leq j \leq b\} \tag{43}$$

Given that X is ordered, if x_i is true, then X has a minimum cardinality of i, so the merge network Y of X and any other input has a minimum

[16] The total number of elements in $|Z_i|$, true or false. This is normally also called cardinality, but this would obviously be ambiguous in our current context where cardinality refers to the number of *true* variables.

cardinality i: so output y_i is true. Same for y_j. If and x_i and x'_j are *both* true, then we have a minimum cardinality of the sum $i+j$, so y_{i+j} is true. Now, constraining $\overline{y_{c+1}}$ will set the maximum cardinality of X to c.

Again, for minimum cardinality (and additionally for exact cardinality), we need to create implications that make the appropriate elements of the output false instead of true, in the same vein as the direct sorting networks:

$$\{\neg x_i \Rightarrow \neg y_{b+i}, \neg x'_j \Rightarrow \neg y_{a+j}, \neg x_i \wedge \neg x'_j \Rightarrow \neg y_{i+j-1} : 1 \leq i \leq a, i \leq j \leq b\} \tag{44}$$

Again, if x_i is false, then X has a maximum cardinality of $i-1$, so the merge network of X and input X' (which could have at most b true variables) has a maximum cardinality of $b+i$: so y_{b+i} must be false. Same for $\overline{y_{a+j}}$. If x_i and x'_j are both false, then we have a maximum cardinality of $i+j-1$, so y_{i+j-1} is false. Now, constraining y_c will set the minimum cardinality of X to c.

- **c-cardinality merge networks, direct method** For this, we can simply skip adding any $\cdots \Rightarrow y_k$ where $k > c$.

Sorting, merge and cardinality networks, mixed method
Finally, the mixed method calculates at each sort or merge step the amount of variables V and clauses C either recursive or direct method will produce, and chooses the method that minimizes some ratio $\lambda V + C$ for a given (λ which we have set to 5).

F Binary Decision Diagrams (BDDs)

A BDD is a binary tree that describes a circuit of *if-then-else* gates. Each node of the tree has one false and one true child. The leaves of the tree are the 1- or 0-terminals, representing the final output of the circuit. A BDD can be constructed to encode any PB constraint. The i'th variable of the PB constraint is represented by the node(s) on depth i. If a path through the tree exists at a 1-terminal, the PB constraint is satisfied if the variables are set to true or false based on whether the path at the variable's tree node goes through the true or false child. See Figure 3 for some examples of PB constraints encoded as BDDs.

Once constructed, a BDD can be easily translated to CNF using the Tseitin transformation (see section F). The size of the final encoding depends on the size of the BDD, but there is no guarantee that the size will be linear. In comparison with sorting networks and binary adders, it has been found that it's the best way to encode PB constraints [38]. Also in comparison with Multi-Decision Diagrams (MDDs), which work well for small and medium sized instances [39], BDDs seem somewhat more consistent and a solid choice.

We have implemented the interval BDD construction procedure [40], which creates more compact (and canonical) ROBDDs (Reduced Order BDDs) by

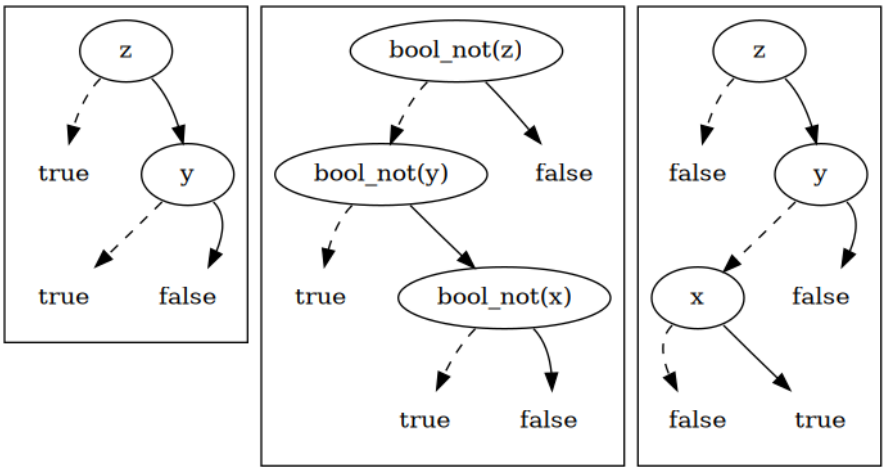


Figure 3: From left to right, these three BDDs encode the following PB constraints: $2x + 3y + 5z \leq 7$, $2x + 3y + 5z \geq 7$ and $2x + 3y + 5z = 7$. The dashed edges are false edges (denoting the edge to the false child), the other are true edges. The 0- and 1-terminals are (in this case) the Boolean literals `false` and `true`, but they could also be swapped (to negate the constraint) or non-fixed variables (to reify the PB constraint). We follow a path from the root to a terminal, and make a decision for each variable at each node. For example, in the first BDD we see that a false assignment of z is sufficient for the constraint to be satisfied, since even if x and y are true, their sum will be $5 \leq 7$. If z is true, we have to assign y false, otherwise we'd get $5 + 3 \not\leq 7$. If y is false, we don't need to check x since $2x \leq 7$ holds no matter the assignment of x, which there are no nodes for x in the left-most BDD. Placing variables with large coefficients at the top often make small decisions later on irrelevant. This visualization was automatically generated by the `sat` library using the DOT graph description language.

merging BDD nodes that represent equivalent PB constraint. For example, $2x_1 + 3x_2 + 5x_3 \leq 6$ and $2x_1 + 3x_2 + 5x_3 \leq 5$ are equivalent constraints, since no assignment of x_1, x_2 and x_3 will add up to 6. So the node for this particular PB constraint can represent multiple values (in fact, an interval) instead of a single value for its right-hand side: in this case $[5, 6]$.

Any comparison operator $\#$ can be rewritten to $\leq$, for instance: $\sum a_i x_i = K \Leftrightarrow \sum a_i x_i \leq K \wedge \sum a_i x_i \geq K$, and $\sum a_i x_i \geq K \Leftrightarrow \sum a_i \overline{x_i} < K \Leftrightarrow \sum a_i \overline{x_i} \leq K - 1$. Thus, the paper only considers the $\leq$ case for their algorithm.

We apply the following pre-processing steps on the PB-constraint before the construction procedure:

- Filter out every fixed variable x_i (subtract a_i from K if x_i is true) and terms where $a_i = 0$.
- Since the algorithm expects positive coefficients, the sign of each negative coefficient is flipped by negating its variable. At this point, with only positive coefficients, if $K < 0$ the PB constraint is UNSAT. If $K = 0$ all variables can be set to false, and no BDD is necessary.
- As noted in the paper [40], ordering the terms by decreasing coefficient value leads generally to smaller BDDs (as more impactfull decisions are made higher up in the tree, leading to a narrower tree).

The aforementioned intervals cannot be simply determined by looking at the whole PB-constraint, but have to be calculated bottom-up (i.e. the interval of a node can be determined from the intervals of its child nodes), which is what the BDD interval construction procedure does. Roughly speaking, the procedure maintains a set of n maps (the *levels*), one for each term of the PB constraint. Level k maps intervals to BDD nodes for a particular sub-constraint of only the last k terms: $\sum_{i=k}^{n} a_i x_i \leq K'$. The variables x_i for $0 \leq i \leq k-1$ are assumed to have been already assigned and their coefficients a_i to have been subtracted from the right-hand side if $x_i = 1$, leaving $K' \leq K$ as the new maximum sum for the rest. At the last level, we have a single inequality constraint for which we can determine the interval, which will percolate up to the caller.

Let's define $M_k = \sum_{i=k}^{n} a_i$, the sum of the (remaining) left-hand side coefficients at level k. No assignment of x will yield a sum greater than M_k. So, at any level k, the sub-constraint might be trivially satisfied if the $M_k \leq K'$ (since any assignment of the remaining x_i will lead to SAT) or if $K' < 0$ (no assignment will lead to SAT, since all coefficients are positive). To terminate in these cases, each level is initialized with two elements, one for SAT holding the 1-terminal $\underline{1}$ and the other for UNSAT with the 0-terminal $\underline{0}$:

$$L_k = \left\{ \Big((-\infty, -1], \underline{0} \Big), \Big([M_k, \infty), \underline{1} \Big) \right\}$$

These two elements the initial base cases for constructing a BDD for $\leq$ PB constraints, and any recursive branch of the algorithm will terminate in these base cases (or in memoized BDD from a level map, for which the same is true). We have found that by changing the initial base cases, we can encode other PB constraints besides $\leq$ (perhaps the authors of [40] were aware of this possibility, but it's not discussed in their paper). For instance, $>$ (the inverse of $\leq$) can naturally be encoded by swapping out the 0 and 1 terminals. More interestingly, the BDD for a $=$ PB constraint can be constructed by adding the following initial base cases for each level k:

$$L_k = \left\{ \Big((-\infty, -1], \underline{0} \Big), \Big([M_k, M_k], \underline{1} \leftrightarrow \bigwedge_{i=k}^{n} x_i \Big), \Big([M_k + 1, \infty), \underline{0} \Big) \right\}$$

Now, arriving at level with a K' less than 0 is still UNSAT, but if it's strictly greater than M_k the sub-constraint is also UNSAT (even if all variables are assigned true, the left-hand side will fall short of the right-hand side). If left- and right-hand side are exactly equal ($M_k = K'$), we assign all remaining variables to be true[17]. The advantage of directly constructing an equality BDD in this way is that we don't need two inequality BDDs. However, less nodes will be merged than in those two as the inner BDD nodes on each level only cover intervals of size 1. In Figure 3, the third BDD has been created with this method, which yields a BDD that is about the same size as either inequality BDD. More study is needed to see what is the best in general (or whether we can predict which approach will be better).

The $\underline{0}$ and $\underline{1}$ terminals are Boolean literals, which means we can assign either $false$ or $true$ to swap them, or assign reification variables $\overline{r}$ or r if we need a reified PB constraint (as we will for `int_lin_le_reif` and `int_lin_eq_reif`).

The most suitable data-structure for the level maps is an interval tree, so that nodes can be stored with an interval key $[\beta, \gamma]$, and retrieved with any point value k that falls within the interval of the key of X. Such a tree is not included in the C++ `std` library, but since `std::map` checks for equivalence of two keys objects `a` and `b` using `!(a < b) && !(b < a)`, we can make a very minimal implementation of an interval tree by overloading the <-operator for two BDD interval `structs`:

```
bool operator<(const BDDInterval &lhs,
    const BDDInterval &rhs) { return lhs.u < rhs.l; };
```

If a map constrains an entry X with interval key $[\beta, \gamma]$, and we do a look-up for point value k for which $\beta \leq k \leq \gamma$ expressed as interval $[k, k]$, then the equivalence check will pass and return X since:

$$\neg([k,k] < [\beta,\gamma]) \wedge \neg([\beta,\gamma] < [k,k]) \Leftrightarrow \neg(k < \beta) \wedge \neg(\gamma < k) \Leftrightarrow \beta \leq k \leq \gamma$$

The BDD is output by the construction as a new `bdd` global, which takes as input the BDD expressed as an array of variables, an array of true child edges and another of false child edges, and its terminal nodes. This can be constructed using breadth-first search from the levels structure, which is implemented for the following new `bdd` global.

`bdd(array[int] of var bool:` X`, array[int] of int:` F`, array[int] of int:` T`, var bool:` z`, var bool:` o`) This BDD global denotes the n nodes of a BDD: each X_i is the Boolean variables of node i (which will have come from the original PB constraint the BDD is encoding, but there will be duplicates), F and T are the false and true edge sets the tree (e.g., $L_2 = 3$ means that the false child of node 2 is node 3), and z and o are the 0 and

[17] Actually, it seems this last base case can be omitted, which is probably more efficient. We didn't have time to adequately test this yet so the final implementation includes all three base cases.

1-terminal, respectively. Because there is no edge to the root of a tree, we can use 0 and 1 to refer to the 0- and 1-terminal, as we don't have to use 1 to refer to node 1, so $L[3] = 1$ unambiguously means that the true child of node 3 is the 1-terminal, not the root.

The BDD can be encoded into CNF with the so-called Tseitin transformation, which creates an array of auxiliary Boolean variables V where each V_i encodes the truth value of the BDD rooted at node i, and 6 clauses per node [40]. It is also possible to reduce this number to 3 clauses per node at the cost of propagation strength, or use a so-called path-based encoding instead of Tseitin [41]. The Tseitin transformation X decide o and z, but if $o = \overline{z} = r$ in the fully reified PB constraint $C_{\text{PB}}(X) \Leftrightarrow r$, we must still add the implication $r \Rightarrow C_{\text{PB}}(X)$. This we can do by adapting the path-based encoding only for the terminals: if terminal o (or z) is true, then at least one of that terminal's parents p must be in the path (which means V_p), and the parent's variable X_p must be true if the terminal is its true child, or false if it is its false child:

$$\begin{aligned}\left(z \Rightarrow \left(\bigvee_{f\in F, f=0} V_f \wedge \overline{X_f}\right) \vee \left(\bigvee_{t\in T, t=0} V_t \wedge X_t\right)\right) \wedge \\ \left(o \Rightarrow \left(\bigvee_{f\in F, f=1} V_f \wedge \overline{X_f}\right) \vee \left(\bigvee_{t\in T, t=1} V_t \wedge X_t\right)\right)\end{aligned} \tag{45}$$

These four disjunction select the false parents of z (the nodes with z as their false child), the true parents of z, the false parent of o and finally the true parents of o, respectively.

G FlatZinc built-ins: arithmetical (plus, times, div) integer constraints

G.1 Dual encodings

For two integer variables, the dual encoding function f introduces (on top of their direct encoding) another encoding for every conjunction of two encoding variables: $f(x = v \wedge y = w) = f(x = v) \wedge f(y = w)$. It does this by creating a matrix B of fresh Boolean variables with dimensions, corresponding to the bounds of x and y: $D_u(x) - D_l(x) + 1 \times D_u(y) - D_l(y) + 1$. Then, $B_{v,w} \Leftrightarrow f(x = v) \wedge f(y = w)$ (in some sense, B acts as a reification matrix). Such a dual encoding is possible for any encoding (or any two arrays of Boolean variables), and also for other Boolean connectives, but we only use with the direct encoding with conjunction.

int_plus(var int: a, var int: b, var int: c)[18] Constrains $a+b=c$ for integer variables. With the direct encoding at least one fixed variable, we have the benefit that simple algebra can be applied as normal. For example, in the constraint $a+2=b$ with $D(a)=\{1,2,3\}$ and $D(b)=\{3,4\}$ we get $f(a=1+2) \Leftrightarrow f(b=3) \wedge f(a=2+2) \Leftrightarrow f(b=4) \wedge f(a=3+2) \Leftrightarrow f(b=5) \Leftrightarrow 0$. For the first two bi-conditionals, unification shares the encoding variables of a and b, and the last one $f(a=3+2)$ becomes fixed. The effect of this unification is that the encoding of b is replaced by the encoding of a with its index set shifted by 2 and with encoding variables of a that once shifted don't coincide with the domain of b pruned ($f(a=3)$ in this case). This means the constraint requires no clauses, and only 2 variables (instead of 5). In general, addition (and subtraction) of fixed values is resolved during compilation in $O(|D(a)|+|D(b)|+|D(c)|)$ time (with one of three terms being 1).

Thus, if at least one of the three variables is fixed, we can resolve the constraint during compilation. If all variables are non-fixed, we have to build a one-way implication between the encoding variables (although two-way would be possible as well and might aid propagation):

$$\begin{cases} \bigwedge_{d_i \in D(a) \cap D(c)} f(a=d_i) \Leftrightarrow f(c=d_i+b) & \text{if } b \text{ is fixed (similar for } a) \\ \bigwedge_{d_i \in D(a) \cap D(b)} f(a=d_i) \Leftrightarrow f(b=c-d_i) & \text{if } c \text{ is fixed} \\ \bigwedge_{d_i \in D(a)} \bigwedge_{d_j \in D(b)} f(a=d_i) \wedge f(b=d_j) \Rightarrow f(c=d_i+d_j) & \text{if non are fixed} \end{cases} \tag{46}$$

Additionally, we added a functional equivalent of the `int_plus` constraint with fixed integer a and integer variable b: it then creates integer variable c, constrains $a+b=c$, and returns c. Again, if either a or b is fixed, no extra variables end up being added to the model, as the direct encoding of c yields the direct encoding of a with its index set *shifted* by b. Accordingly, $D(c)=\{a+d_i : d_i \in D(b)\}$.

int_times(var int: a, var int: b, var int: c) Constrains $ab=c$ for integer variables. Again, if at least one variable is fixed, simple algebra can take care of this constraint:

$$\begin{cases} \bigwedge_{d_i \in D(a) \cap D(b)} f(a=d_i) \Leftrightarrow f(c=d_i b) & \text{if } b \text{ is fixed} \\ \bigwedge_{d_i \in D(a)} \bigwedge_{d_j \in D(b)} \begin{cases} f(a=d_i) \Leftrightarrow f(b=d_j) & \text{if } d_i d_j = c \\ f(a \neq d_i) \vee f(b \neq d_j) & \text{otherwise} \end{cases} & \text{if } c \text{ is fixed} \end{cases} \tag{47}$$

[18] The following arithmetical FlatZinc built-ins (`int_plus`, `int_times`, `int_pow`, `int_div`) are implemented for most encodings, but are not evaluated in the experiments because of two reasons. First, in the `plus` case, the compiler uses `int_lin_eq` constraints instead (which uses PBLib). For some cases (as explained in this section), it would be beneficial to use `int_plus`, but we don't have full confidence in the implementation for the order and binary encoding yet, so we stick to `int_lin_eq` in this version. Secondly, the other three arithmetical constraints are not used in the set of models in the evaluation. So, while all following constraints are unit tested, and would work for user models that use them, they don't contribute to the results in section 5.

In general, multiplication of fixed values is resolved during compilation in $O(|D(a)||D(b)||D(c)|)$ time (with one of the three terms being 1).

Like for the `int_plus` constraints, we added a functional equivalent of `int_times` with fixed integer a and one integer variable b, which will create integer variable c, constrain $ab = c$, and return c. Thus, the direct encoding of c yields the direct encoding of a with its index set *scaled* by b. Accordingly, $D(c) = \{ad_i : d_i \in D(b)\}$. Between each subsequent d_i, gaps will appear in $D(c)$ of size $a - 1$, which will slow down flattening because for every direct encoding $f(c = v)$ we need to check whether $v \in D(c)$. So, another approach is to use an interval domain $D(c) = [D_l(ab)..D_u(ab)]$, but this will increase the amount of variables.

If all variables are non-fixed, we mix the approach from the `linear` library [11], the `array_int_element`, and the dual encoding introduced in section G.1:

$$\begin{cases} (f(a=0) \vee f(b=0) \Leftrightarrow f(c=0)) \wedge (f(a=i) \vee f(b=j) \Leftrightarrow f(c=ij)) & \text{case 1} \\ f(c = (f(b = d_i b) : d_i \in D(a))_a) & \text{if } |D(c)| > 20 \\ \bigwedge_{d_i \in D(a) \wedge d_j \in D(b)} f(a = d_i, b = d_j) \Leftrightarrow f(c = d_i d_j) & \text{otherwise} \end{cases} \tag{48}$$

Where case 1 is if $D(a) = \{0, i\} \wedge D(b) = \{0, j\}$

int_div This implementation was adapted from the linear library [11].

H Constraining Boolean cardinality constraints for a non-fixed count c

Referring back to the Boolean cardinality constraint $\sum Y = c$ in section 4.2.5, we present an alternate approach by constructing a sorting network in the case that c is non-fixed.

Let $c' = \min\{|Y|, D_u(c)\}$ if minimum, or $c' = \ldots + 1$ if maximum or exact cardinality: we don't have to count more variables than the upper bound of c allows, and it is impossible to have a sorting network larger than the amount of variables in Y. Then, we construct a c'-cardinality network $S = (s_1, s_2, \ldots, s_{c'})$ of Y. Since S is decreasing and $\sum S = \sum Y$, S is equivalent to the order encoding for $\sum Y$:

$$\bigwedge_{d_i \in D(c)} f(c \geq d_i) \Leftrightarrow s_{d_i} \begin{cases} s_{d_i} & \text{if } d_i \leq c' \\ 0 & \text{otherwise} \end{cases} \tag{49}$$

I The `--dimacs-idn-mode` flag

The DIMACS-IDN mode interface flag tries to match the IDN numbers of FlatZinc variables (42 for variable `X_INTRODUCED_42`) with the DIMACS ID's (`42`)

of the solver input model for improved debugability and theoretically better performance. By having this direct link, the two-way map between FlatZinc, in-memory model and DIMACS file doesn't have to be implemented with a map, as the DIMACS ID's can now be directly inferred from the IDN and back. This mode worked for some models, but was never fully finished.

The challenges for this mode were that firstly, `IDN`'s start at 0, while DIMACS IDS start at 1, so the match would always be off-by-one. Next, `IDN`'s are non-continuous, so the set variable amount in the `p-line` would be higher than necessary, which leads to free, unrestricted variables for the SAT-solver. A DIMACS model using two variables with IDs 1 and 42 would have 42 variables. It depends on the solver whether the free variables 2-41 impact performance. Lastly, variables that left over from the original model retain their original ID, and don't have an IDN number. These would have to be still mapped to a number with a map data-structure.

J Evaluation of different `mzn-sat` default encodings and backend Max-SAT solvers

To select the best solver and see if there is any connection between solvers and encodings, we will test each default encoding setting with the three different Max-SAT solvers we selected based on the result from the Max-SAT 2019 competition. First, `rc2` [42] is a core-guided Max-SAT solver which won the weighted and unweighted categories of the main track in the Max-SAT 2019 competition. This is a complete solver, which means it only tries to find optimal solutions. Second, `tt-open-wbo-inc` which won first place for the weighed categories on the incomplete track, and `loandra`, which took first place for unweighted and second place for weighted in the incomplete track.

The experiments were run with a 20 minute timeout on a single-core *Intel Xeon 8260* CPU (non-hyperthreaded) with 4 GB of RAM made available, on a compute server courtesy of Monash University.

Table 7: Aggregate counts of status output by MiniZinc for six models, each with five instances. Compares three SAT solvers and three default encodings, where `opt` indicates that optimality was proven, `sat` feasible without optimality proven, `unk` for timeout reached before a feasible solution was found and `err` for out-of-memory related error.

	loandra				rc2				tt-open-wbo-inc			
direct	opt	sat	unk	err	opt	sat	unk	err	opt	sat	unk	err
fox-geese-corn	1	1	0	3	1	0	2	2	2	1	0	2
hrc	5	0	0	0	5	0	0	0	5	0	0	0
lot-sizing	0	5	0	0	0	0	5	0	0	5	0	0
median-string	3	1	0	1	0	0	4	1	2	2	0	1
multi-knapsack	0	0	0	5	0	0	0	5	0	0	0	5
triangular	1	4	0	0	1	0	4	0	1	4	0	0
Totals:	10	11	0	9	7	0	15	8	10	12	0	8
order	opt	sat	unk	err	opt	sat	unk	err	opt	sat	unk	err
fox-geese-corn	2	0	0	3	1	0	2	2	2	1	0	2
hrc	5	0	0	0	5	0	0	0	5	0	0	0
lot-sizing	0	5	0	0	0	0	5	0	0	5	0	0
median-string	3	1	0	1	2	0	2	1	2	2	0	1
multi-knapsack	0	0	0	5	0	0	0	5	0	0	0	5
triangular	1	4	0	0	1	0	4	0	1	4	0	0
Totals:	11	10	0	9	9	0	13	8	10	12	0	8
binary	opt	sat	unk	err	opt	sat	unk	err	opt	sat	unk	err
fox-geese-corn	1	2	0	2	1	0	2	2	2	1	0	2
hrc	5	0	0	0	5	0	0	0	5	0	0	0
lot-sizing	0	2	0	3	0	0	2	3	0	2	0	3
median-string	2	2	0	1	2	0	2	1	2	2	0	1
multi-knapsack	1	0	0	4	1	0	0	4	1	0	0	4
triangular	1	4	0	0	1	0	4	0	1	4	0	0
Totals:	10	10	0	10	10	0	10	10	11	9	0	10

Looking at the result between the solvers in Table 7, the most pronounced difference is between the complete solver `rc2` and the other incomplete solvers. As expected, `sat` is not reached for the complete solver, but the count of `opt` is also significantly lower: especially for direct, and somewhat for order encoding. Perhaps `rc2` does not scale well for problems with a large number of variables. However, it should still be retried in the future for compact encodings like binary.

From this table, we determine that `tt-open-wbo-inc` is a good choice for our comparison with the control solvers. While it has one less `opt` status, it also has one less `err`, so we will go with consistency. It does not seem that any particular solver benefits from a specific encoding more than any other solver, but this is hard to tell.

K Per model results

Table 8: Per-model, aggregate results from four test (`mzn-sat` with different default encoding configurations) and four control solvers solving the MiniZinc Challenge 2019, which counts 20 models, each with five instances. The first column shows model name, the second the configuration tested. The next four columns show the breakdown of statuses output for the instances: the `opt` status indicates that optimality was proven on the best found solution, `sat` that the instance was feasible but without optimality proven, `uns` that instance is unsatisfiable, `unk` for timeout reached before a feasible solution was found and `err` that there was an error of some kind. Then, the `inc` column shows how many of the `opt`, `sat` and `uns` statuses were incorrect, and the `fail` column the amount of instances where the solver failed to find a correct solution (which is the sum of the previous three columns).

configuration	problem	opt	sat	uns	unk	err	inc	fail
mzn-sat-mixed	accap	0	1	0	0	4	0	4
mzn-sat-direct	accap	0	0	0	0	5	0	5
mzn-sat-order	accap	0	1	0	0	4	0	4
mzn-sat-binary	accap	2	1	0	0	2	0	2
gecode	accap	0	5	0	0	0	0	0
gurobi	accap	1	2	0	2	0	0	2
chuffed	accap	3	2	0	0	0	0	0
picat-sat	accap	3	2	0	0	0	0	0
mzn-sat-mixed	amaze	0	5	0	0	0	0	0
mzn-sat-direct	amaze	0	4	0	1	0	0	1
mzn-sat-order	amaze	0	4	0	0	1	0	1
mzn-sat-binary	amaze	0	5	0	0	0	0	0
gecode	amaze	0	5	0	0	0	0	0
gurobi	amaze	0	4	0	1	0	0	1
chuffed	amaze	0	5	0	0	0	0	0

picat-sat	amaze	0	5	0	0	0	0	0
mzn-sat-mixed	code-generator	0	0	0	0	5	0	5
mzn-sat-direct	code-generator	0	0	0	0	5	0	5
mzn-sat-order	code-generator	0	0	0	0	5	0	5
mzn-sat-binary	code-generator	0	0	0	0	5	0	5
gecode	code-generator	0	3	0	2	0	0	2
gurobi	code-generator	0	0	2	0	3	2	5
chuffed	code-generator	2	1	0	2	0	0	2
picat-sat	code-generator	0	0	0	5	0	0	5
mzn-sat-mixed	fox-geese-corn	1	2	0	2	0	0	2
mzn-sat-direct	fox-geese-corn	2	1	0	2	0	0	2
mzn-sat-order	fox-geese-corn	2	3	0	0	0	0	0
mzn-sat-binary	fox-geese-corn	4	1	0	0	0	0	0
gecode	fox-geese-corn	1	4	0	0	0	0	0
gurobi	fox-geese-corn	4	1	0	0	0	0	0
chuffed	fox-geese-corn	4	1	0	0	0	0	0
picat-sat	fox-geese-corn	4	0	0	1	0	0	1
mzn-sat-mixed	groupsplitter	0	0	0	0	5	0	5
mzn-sat-direct	groupsplitter	0	0	0	0	5	0	5
mzn-sat-order	groupsplitter	0	0	0	0	5	0	5
mzn-sat-binary	groupsplitter	0	0	0	0	5	0	5
gecode	groupsplitter	2	3	0	0	0	0	0
gurobi	groupsplitter	0	0	0	0	5	0	5
chuffed	groupsplitter	4	1	0	0	0	0	0
picat-sat	groupsplitter	0	0	0	4	1	0	5
mzn-sat-mixed	hrc	5	0	0	0	0	0	0
mzn-sat-direct	hrc	5	0	0	0	0	0	0
mzn-sat-order	hrc	5	0	0	0	0	0	0
mzn-sat-binary	hrc	5	0	0	0	0	0	0
gecode	hrc	1	1	0	3	0	0	3
gurobi	hrc	5	0	0	0	0	0	0
chuffed	hrc	5	0	0	0	0	0	0
picat-sat	hrc	5	0	0	0	0	0	0
mzn-sat-mixed	kidney-exchange	2	3	0	0	0	0	0
mzn-sat-direct	kidney-exchange	2	3	0	0	0	0	0
mzn-sat-order	kidney-exchange	5	0	0	0	0	0	0
mzn-sat-binary	kidney-exchange	5	0	0	0	0	0	0
gecode	kidney-exchange	4	0	0	1	0	0	1
gurobi	kidney-exchange	5	0	0	0	0	0	0
chuffed	kidney-exchange	5	0	0	0	0	0	0
picat-sat	kidney-exchange	5	0	0	0	0	0	0
mzn-sat-mixed	liner-sf-repositioning	0	0	0	2	3	0	5
mzn-sat-direct	liner-sf-repositioning	0	0	0	2	3	0	5
mzn-sat-order	liner-sf-repositioning	0	0	0	0	5	0	5
mzn-sat-binary	liner-sf-repositioning	0	0	0	0	5	0	5
gecode	liner-sf-repositioning	3	1	0	1	0	0	1

gurobi	liner-sf-repositioning	4	1	0	0	0	0	0
chuffed	liner-sf-repositioning	3	1	0	1	0	0	1
picat-sat	liner-sf-repositioning	0	0	0	2	3	0	5
mzn-sat-mixed	lot-sizing	0	1	0	2	2	0	4
mzn-sat-direct	lot-sizing	0	1	0	2	2	0	4
mzn-sat-order	lot-sizing	0	5	0	0	0	0	0
mzn-sat-binary	lot-sizing	1	4	0	0	0	0	0
gecode	lot-sizing	3	2	0	0	0	0	0
gurobi	lot-sizing	3	2	0	0	0	0	0
chuffed	lot-sizing	2	3	0	0	0	0	0
picat-sat	lot-sizing	3	2	0	0	0	0	0
mzn-sat-mixed	median-string	0	3	0	1	1	0	2
mzn-sat-direct	median-string	0	3	0	1	1	0	2
mzn-sat-order	median-string	2	2	0	0	1	0	1
mzn-sat-binary	median-string	2	2	0	1	0	0	1
gecode	median-string	2	3	0	0	0	0	0
gurobi	median-string	0	5	0	0	0	0	0
chuffed	median-string	3	2	0	0	0	0	0
picat-sat	median-string	2	3	0	0	0	0	0
mzn-sat-mixed	multi-knapsack	0	0	0	3	2	0	5
mzn-sat-direct	multi-knapsack	0	0	0	3	2	0	5
mzn-sat-order	multi-knapsack	1	0	0	2	2	0	4
mzn-sat-binary	multi-knapsack	1	0	0	4	0	0	4
gecode	multi-knapsack	2	1	0	2	0	0	2
gurobi	multi-knapsack	5	0	0	0	0	0	0
chuffed	multi-knapsack	3	0	0	2	0	0	2
picat-sat	multi-knapsack	1	1	0	3	0	0	3
mzn-sat-mixed	nside	0	0	0	0	5	0	5
mzn-sat-direct	nside	0	0	0	0	5	0	5
mzn-sat-order	nside	0	0	0	0	5	0	5
mzn-sat-binary	nside	0	0	0	0	5	0	5
gecode	nside	0	4	0	0	1	0	1
gurobi	nside	0	0	3	1	1	3	5
chuffed	nside	3	1	0	0	1	0	1
picat-sat	nside	0	0	0	3	2	0	5
mzn-sat-mixed	ptv	0	0	0	0	5	0	5
mzn-sat-direct	ptv	0	0	0	0	5	0	5
mzn-sat-order	ptv	0	0	0	4	1	0	5
mzn-sat-binary	ptv	2	1	0	1	1	0	2
gecode	ptv	0	5	0	0	0	0	0
gurobi	ptv	5	0	0	0	0	0	0
chuffed	ptv	3	2	0	0	0	0	0
picat-sat	ptv	3	2	0	0	0	0	0
mzn-sat-mixed	rcpsp-wet-diverse	0	0	0	0	5	0	5
mzn-sat-direct	rcpsp-wet-diverse	0	0	0	0	5	0	5
mzn-sat-order	rcpsp-wet-diverse	0	0	0	0	5	0	5

mzn-sat-binary	rcpsp-wet-diverse	0	0	0	0	5	0	5
gecode	rcpsp-wet-diverse	0	1	0	3	1	0	4
gurobi	rcpsp-wet-diverse	1	1	0	2	1	0	3
chuffed	rcpsp-wet-diverse	2	2	0	0	1	0	1
picat-sat	rcpsp-wet-diverse	0	2	0	2	1	0	3
mzn-sat-mixed	rotating-workforce	0	0	0	2	3	0	5
mzn-sat-direct	rotating-workforce	0	0	0	2	3	0	5
mzn-sat-order	rotating-workforce	0	0	0	3	2	0	5
mzn-sat-binary	rotating-workforce	0	2	0	3	0	0	3
gecode	rotating-workforce	0	1	0	4	0	0	4
gurobi	rotating-workforce	0	3	2	0	0	0	0
chuffed	rotating-workforce	0	3	1	1	0	0	1
picat-sat	rotating-workforce	0	2	2	1	0	0	1
mzn-sat-mixed	stack-cuttingstock	1	1	0	0	3	0	3
mzn-sat-direct	stack-cuttingstock	0	0	0	0	5	0	5
mzn-sat-order	stack-cuttingstock	1	1	0	0	3	0	3
mzn-sat-binary	stack-cuttingstock	1	4	0	0	0	0	0
gecode	stack-cuttingstock	1	2	0	2	0	0	2
gurobi	stack-cuttingstock	5	0	0	0	0	0	0
chuffed	stack-cuttingstock	1	4	0	0	0	0	0
picat-sat	stack-cuttingstock	3	2	0	0	0	0	0
mzn-sat-mixed	steelmillslab	0	0	0	0	5	0	5
mzn-sat-direct	steelmillslab	0	0	0	0	5	0	5
mzn-sat-order	steelmillslab	0	0	0	0	5	0	5
mzn-sat-binary	steelmillslab	0	0	0	0	5	0	5
gecode	steelmillslab	3	2	0	0	0	0	0
gurobi	steelmillslab	3	1	0	1	0	0	1
chuffed	steelmillslab	3	2	0	0	0	0	0
picat-sat	steelmillslab	0	5	0	0	0	0	0
mzn-sat-mixed	stochastic-vrp	4	1	0	0	0	0	0
mzn-sat-direct	stochastic-vrp	3	2	0	0	0	0	0
mzn-sat-order	stochastic-vrp	5	0	0	0	0	0	0
mzn-sat-binary	stochastic-vrp	5	0	0	0	0	0	0
gecode	stochastic-vrp	3	2	0	0	0	0	0
gurobi	stochastic-vrp	1	4	0	0	0	0	0
chuffed	stochastic-vrp	5	0	0	0	0	0	0
picat-sat	stochastic-vrp	5	0	0	0	0	0	0
mzn-sat-mixed	triangular	1	4	0	0	0	0	0
mzn-sat-direct	triangular	1	4	0	0	0	0	0
mzn-sat-order	triangular	1	4	0	0	0	0	0
mzn-sat-binary	triangular	1	4	0	0	0	0	0
gecode	triangular	1	4	0	0	0	0	0
gurobi	triangular	1	4	0	0	0	0	0
chuffed	triangular	0	5	0	0	0	0	0
picat-sat	triangular	1	0	0	4	0	0	4
mzn-sat-mixed	zephyrus	0	0	0	0	5	0	5

mzn-sat-direct	zephyrus	0	0	0	0	5	0	5
mzn-sat-order	zephyrus	0	0	0	0	5	0	5
mzn-sat-binary	zephyrus	5	0	0	0	0	0	0
gecode	zephyrus	3	0	0	2	0	0	2
gurobi	zephyrus	5	0	0	0	0	0	0
chuffed	zephyrus	5	0	0	0	0	0	0
picat-sat	zephyrus	5	0	0	0	0	0	0

References

[1] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[2] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[4] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

[5] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[6] Jessica Davies. *Solving MaxSAT by decoupling optimization and satisfaction*. PhD thesis, University of Toronto, 2013.

[7] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-wbo: A modular maxsat solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014.

[8] Saurabh Joshi, Prateek Kumar, Sukrut Rao, and Ruben Martins. Open-wbo-inc: Approximation strategies for incomplete weighted maxsat. *J. Satisf. Boolean Model. Comput.*, 11(1):73–97, 2019.

[9] Inês Lynce and Joël Ouaknine. Sudoku as a SAT problem. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2006, Fort Lauderdale, Florida, USA, January 4-6, 2006*, 2006.

[10] Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

[11] Gleb Belov, Peter J Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In *International Conference on Principles and Practice of Constraint Programming*, pages 49–65. Springer, 2016.

[12] Jinbo Huang. Universal booleanization of constraint models. In *International Conference on Principles and Practice of Constraint Programming*, pages 144–158. Springer, 2008.

[13] Peter J. Stuckey, Maria J. García de la Banda, Michael J. Maher, Kim Marriott, John K. Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The G12 project: Mapping solver independent models to efficient solutions. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 13–16. Springer, 2005.

[14] Naoyuki Tamura and Mutsunori Banbara. Sugar: A csp to sat translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.

[15] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.

[16] Justyna Petke and Peter Jeavons. The order encoding: From tractable csp to tractable sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 371–372. Springer, 2011.

[17] Naoyuki Tamura, Mutsunori Banbara, and Takehide Soh. Compiling pseudo-boolean constraints to SAT with order encoding. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 1020–1027. IEEE Computer Society, 2013.

[18] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient sat-encoding of finite domain CSP. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 375–376. Springer, 2011.

[19] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In *International Symposium on Practical Aspects of Declarative Languages*, pages 48–62. Springer, 2016.

[20] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat* . Springer Briefs in Intelligent Systems. Springer, 2015.

[21] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints: An International Journal*, 14(3):357–391, 2009.

[22] Pedro Barahona, Steffen Hölldobler, and Van-Hau Nguyen. Efficient sat-encoding of linear CSP constraints. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2014, Fort Lauderdale, FL, USA, January 6-8, 2014*, 2014.

[23] Hamed Gorjiara, GUOQING HARRY XU, and BRIAN DEMSKY. *Satune: Synthesizing Efficient SAT Encoders*. PhD thesis, UC Irvine, 2019.

[24] Ian P. Gent. Arc consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[25] Magnus Björk. Successful SAT encoding techniques. *J. Satisf. Boolean Model. Comput.*, 7(4):189–201, 2011.

[26] Marco Gavanelli. The log-support encoding of CSP into SAT. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2007.

[27] John von Neumann. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, 1993.

[28] Peter J. Stuckey and Guido Tack. Minizinc with functions. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2013.

[29] Anthony Silvestre. Solving NP-hard problems using quantum computing. *(Unpublished BSc. thesis at Monash University, Australia)*, 2018.

[30] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[31] Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half reification and flattening. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2011.

[32] Gilles Audemard, Loïc Paulevé, and Laurent Simon. SAT heritage: A community-driven effort for archiving, building and running more than thousand SAT solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International*

Conference, Alghero, Italy, July 3-10, 2020, Proceedings, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113. Springer, 2020.

[33] Tobias Philipp and Peter Steinke. Pblib – a library for encoding pseudo-boolean constraints into cnf. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer International Publishing, 2015.

[34] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[35] Van-Hau Nguyen and Son T Mai. A new method to encode the at-most-one constraint into sat. In *Proceedings of the Sixth International Symposium on Information and Communication Technology*, pages 46–53, 2015.

[36] Jingchao Chen. A new sat encoding of the at-most-one constraint. *Proceedings of Constraint Modelling and Reformulation*, 2010.

[37] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In Christian Schulte, editor, *Proceedings of Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013*, volume 8124 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2013.

[38] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[39] Ignasi Abío and Peter J Stuckey. Encoding linear constraints into sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[40] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012.

[41] Ignasi Abío, Graeme Gange, Valentin Mayer-Eichberger, and Peter J Stuckey. On cnf encodings of decision diagrams. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 1–17. Springer, 2016.

[42] Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.