An Introduction to MiniZinc
Version 1.2

Kim Marriott Peter J. Stuckey Leslie De Koninck
Horst Samulowitz

Contents

1 Introduction

MiniZinc is a language designed for specifying constrained optimization and decision problems over integers and real numbers. A MiniZinc model does not dictate how to solve the problem although the model can contain annotations which are used to guide the underlying solver.

MiniZinc is designed to interface easily to different backend solvers. It does this by transforming an input MiniZinc model and data file into a FlatZinc model. FlatZinc models consist of variable declaration and constraint definitions as well as a definition of the objective function if the problem is an optimization problem. The translation from MiniZinc to FlatZinc is specializable to individual backend solvers, so they can control what form constraints end up in. In particular, MiniZinc allows the specification of global constraints by decomposition.

2 Basic Modelling in MiniZinc

In this section we introduce the basic structure of a MiniZinc model using two simple examples.

2.1 Our First Example

Figure 1: Australian states.

As our first example, imagine that we wish to colour a map of Australia as shown in Figure 1. It is made up of seven different states and territories each of which must be given a colour so that adjacent regions have different colours.

We can model this problem very easily in MiniZinc . The model is shown in Figure 2. The first line in the model is a comment. A comment starts with a '%'

aust.mzn

```
% Colouring Australia using nc colours
int: nc = 3;

var 1..nc: wa;   var 1..nc: nt;  var 1..nc: sa;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;   var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

solve satisfy;

output ["wa=", show(wa), "\t nt=", show(nt),   "\t sa=", show(sa), "\n",
         "q=", show(q),  "\t nsw=", show(nsw), "\t v=", show(v), "\n",
         "t=", show(t),  "\n"];
```

Figure 2: A MiniZinc model `aust.mzn` for colouring the states and territories in Australia.

which indicates that the rest of the line is a comment. MiniZinc has no begin/end comment symbols (such as C's /* and */ comments).

The next part of the model declares the variables in the model. The line

```
int: nc = 3;
```

specifies a *parameter* in the problem which is the number of colours to be used. Parameters are similar to variables in most programming languages. They must be declared and given a type. In this case the type is `int`. They are given a value by an *assignment*. MiniZinc allows the assignment to be included as part of the declaration (as in the line above) or to be a separate assignment statement. Thus the following is equivalent to the single line above

```
int: nc;
nc = 3;
```

Unlike variables in many programming languages a parameter can only be given a *single* value. It is an error for a parameter to occur in more than one assignment.

The basic parameter types are integers (`int`), floating point numbers (`float`), Booleans (`bool`) and strings (`string`). Arrays and sets are also supported.

MiniZinc models can also contain another kind of variable called a *decision variable*. Decision variables are variables in the sense of mathematical or logical variables. Unlike parameters and variables in a standard programming language, the modeller does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is.

In our example model we associate a *decision variable* with each region, wa, nt, sa, q, nsw, v and t, which stands for the (unknown) colour to be used to fill the region.

For each decision variable we need to give the set of possible values the variable can take. This is called the variable's *domain*. This can be given as part of the variable declaration and the type of the decision variable is inferred from the type of the values in the domain.

In MiniZinc decision variables can be Booleans, integers, floating point numbers, or sets. Also supported are arrays whose elements are decision variables. In our MiniZinc model we use integers to model the different colours. Thus each of our decision variables is declared to have the domain `1..nc` which is an integer range expression indicating the set $\{1, 2, ..., nc\}$. The type of the values is integer so all of the variables in the model are integer decision variables.

> Identifiers:
> Identifiers which are used to name parameters and variables are sequences of lower and uppercase alphabetic characters, digits and the underscore '_' character. They must start with a alphabetic character. Thus `myName_2` is a valid identifier. MiniZinc (and Zinc) *keywords* are not allowed to be used as identifier names, they are listed in Appendix A. Neither are MiniZinc *operators* allowed to be used as identifier names.

MiniZinc carefully distinguishes between the two kinds of model variables: parameters and decision variables. The kinds of expressions that can be constructed using decision variables are more restricted than those that can be built from parameters. However, in any place that a decision variable can be used, so can a parameter of the same type.

Formally the distinction between parameters and decision variables is called the *instantiation* of the variable. The combination of variable instantiation and type is called a *type-inst*. As you start to use MiniZinc you will undoubtedly see examples of *type-inst* errors.

The next component of the model are the *constraints*. These specify the Boolean expressions that the decision variables must satisfy to be a valid solution to the model. In this case we have a number of not equal constraints between the decision variables enforcing that if two states are adjacent then they must have different colours.

> Relational Operators:
> MiniZinc provides the relational operators: equal (`=`), not equal (`!=`), strictly less than (`<`), strictly greater than (`>`), less than or equal to (`<=`), and greater than or equal to (`>=`).

The next line in the model:

```
solve satisfy;
```

Indicates the kind of problem it is. In this case it is a *satisfaction* problem: we wish to find a value for the decision variables that satisfies the constraints but we do not care which one.

The final part of the model is the *output* statement. This tells MiniZinc what to print when the model has been run and a solution is found.

> Output:
> An output statement is followed by a *list* of strings. These are typically either string literals which are written between double quotes and use a C like notation for special characters, or an expression of the form *show(X)* where X is the name of a decision variable or parameter. In the example `\n` represents the newline character and `\t` a tab.
> String literals must fit on a single line. Longer string literals can be split across multiple lines using the string concatenation operator `++`. For example, the string literal `"Invalid datafile: Amount of flour is non-negative"` is equivalent to the string literal expression `"Invalid datafile: " ++ "Amount of flour is non-negative"`.
> A model can contain at most one output statement.

With the G12 implementation of MiniZinc we can evaluate our model by typing

```
$ mzn aust.mzn
```

where `aust.mzn` is the name of the file containing our MiniZinc model. We must use the file extension ".mzn" to indicate a MiniZinc model.

When we run this we obtain the result:

```
wa=1 nt=3 sa=2
q=1 nsw=3 v=1
t=1
----------
```

The line of 10 dashes `----------` is output automatically added by the MiniZinc output to indicate a solution has been found.

cakes.mzn

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b  <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = ", show(b), "\n",
        "no. of chocolate cakes = ", show(c), "\n"];
```

Figure 3: Model for determining how many banana and chocolate cakes to bake for the school fete.

2.2 An Arithmetic Optimisation Example

Our second example is motivated by the need to bake some cakes for a fete at our local school. We know how to make two sorts of cakes.[1] A banana cake which takes 250g of self-raising flour, 2 mashed bananas, 75g sugar and 100g of butter, and a chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter. We can sell a chocolate cake for $4.50 and a banana cake for $4.00. And we have 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa. The question is how many of each sort of cake should we bake for the fete to maximise the profit. A possible MiniZinc model is shown in Figure 3.

The first new feature is the use of *arithmetic expressions*.

Integer arithmetic Operators:
MiniZinc provides the standard integer arithmetic operators. Addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`div`) and integer modulus (`mod`). It also provides `+` and `-` as unary operators.
Integer modulus is defined to give a result (a `mod` b) that has the same sign as the dividend a. Integer division is defined so that $a = b$`*`(a `div` b) + (a `mod` b). MiniZinc provides standard integer functions for absolute value (`abs`) and power function (`pow`). For example `abs(-4)` and `pow(2,5)` evaluate to 4 and 32 respectively.
The syntax for arithmetic literals is reasonably standard. Integer literals can be decimal, hexadecimal or octal. For instance `0`, `005`, `123`, `0x1b7`, `0o777`.

The second new feature shown in the example is optimisation. The line

[1] WARNING: please don't use these recipes at home

```
solve maximize 400 * b + 450 * c;
```

specifies that we want to find a solution that maximises the expression in the solve statement called the *objective*. The objective can be any kind of arithmetic expression. One can replace the key word *maximize* by *minimize* to specify a minimisation problem.

When we run this we obtain the result:

```
no.  of banana cakes = 2
no.  of chocolate cakes = 2
----------
```

2.3 Datafiles and Assertions

A drawback of this model is that if we wish to solve a similar problem the next time we need to bake cakes for the school (which is often) we need to modify the constraints in the model to reflect the ingredients that we have in the pantry. If we want to reuse the model then we would be better off to make the amount of each ingredient a parameter of the model and then set their values at the top of the model.

Even better would be to set the value of these parameters in a separate *data file*. MiniZinc (like most other modelling languages) allows the use of data files to set the value of parameters declared in the original model. This allows the same model to be easily used with different data by running it with different data files.

Data files must have the file extension ".dzn" to indicate a MiniZinc data file and a model can be run with any number of data files (though a variable/parameter can only be assigned a value in one file.

Our new model is shown in Figure 4. We can run it using the command

```
$ mzn cakes2.mzn pantry.dzn
```

where the data file `pantry.dzn` is defined in Figure 7 gives the same result as `cakes.mzn`. The output from running the command

```
$ mzn cakes2.mzn pantry2.dzn
```

with an alternate data set defined in Figure 7 is

```
no.  of banana cakes = 3
no.  of chocolate cakes = 8
```

Small data files can be entered without directly creating a `.dzn` file, using the command line flag `-D` *string*, where *string* is the contents of the data file. For example the command

```
$ mzn cakes2.mzn -D "flour=4000;banana=6;sugar=2000;butter=500;cocoa=500;"
```

will give indentical results to `mzn cakes2.mzn pantry.dzn`.

Data files can only contain assignment statements which should be to variables and parameters in a model.

Defensive programming suggests that we should check that the values in the data file are reasonable. For our example it is sensible to check that the quantity of all ingredients is non-negative and generate a run-time error if this is not true. MiniZinc provides a built-in Boolean operator for checking parameter values. The form is *assert(B,S)*. The Boolean expression s is evaluated and if it is false execution aborts and the string expression S is evaluated and printed as an error message. To check and generate an appropriate error message if the amount of flour is negative we can simply add the line

```
constraint assert(flour >= 0.0,"Amount of flour is non-negative");
```

to our model. Notice that the *assert* expression is a Boolean expression and so is regarded as a type of constraint. We can add similar lines to check that the quantity of the other ingredients is non-negative.

cakes2.mzn

```
% Baking cakes for the school fete (with data file)

int: flour;  %no. grams of flour available
int: banana; %no. of bananas available
int: sugar;  %no. grams of sugar available
int: butter; %no. grams of butter available
int: cocoa;  %no. grams of cocoa available

constraint assert(flour >= 0,"Invalid datafile: " ++
                  "Amount of flour is non-negative");
constraint assert(banana >= 0,"Invalid datafile: " ++
                  "Amount of banana is non-negative");
constraint assert(sugar >= 0,"Invalid datafile: " ++
                  "Amount of sugar is non-negative");
constraint assert(butter >= 0,"Invalid datafile: " ++
                  "Amount of butter is non-negative");
constraint assert(cocoa >= 0,"Invalid datafile: " ++
                  "Amount of cocoa is non-negative");

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= flour;
% bananas
constraint 2*b  <= banana;
% sugar
constraint 75*b + 150*c <= sugar;
% butter
constraint 100*b + 150*c <= butter;
% cocoa
constraint 75*c <= cocoa;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = ", show(b), "\n",
        "no. of chocolate cakes = ", show(c), "\n"];
```

Figure 4: Data independent model for determining how many banana and c hocolate cakes to bake for the school fete.

2.4 Real Number Solving

MiniZinc also supports "real number" constraint solving using floating point solving. Consider a problem of taking out a short loan for one year to be repaid in 4 quarterly instalments. A model for this is shown in Figure 6. It uses a simple interest calculation to calculate the balance after each quarter.

Note that we declare a float variable f using `var float:` f, and we can declare a float variable f in a fixed range l to u with a declaration of the form `var` l `..` u `:` f, where l and u are floating point expressions.

We can use the same model to answer a number of different questions. The first question is: if I borrow \$1000 at 4% and repay \$260 per quarter, how much do I end up owing? This question is encoded by the data file `loan1.dzn`.

Since we wish to use real number solving we need to use a different solver than

pantry.dzn

```
flour = 4000;
banana = 6;
sugar = 2000;
butter = 500;
cocoa = 500;
```

pantry2.dzn

```
flour = 8000;
banana = 11;
sugar = 3000;
butter = 1500;
cocoa = 800;
```

Figure 5: Example data files for `cakes2.mzn`

loan.mzn

```
% variables
var float: R;        % quarterly repayment
var float: P;        % principal initially borrowed
var 0.0 .. 100.0: I; % interest rate

% intermediate variables
var float: B1; % balance after one quarter
var float: B2; % balance after two quarters
var float: B3; % balance after three quarters
var float: B4; % balance owing at end

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output ["Borrowing ", show(P), " at ", show(I*100.0), "% interest, ",
        "and repaying ", show(R), "\nper quarter for 1 year leaves ",
        show(B4), " owing\n"];
```

Figure 6: Model for determining relationships between a 1 year loan repaying every quarter.

the default finite domain solver used by `mzn`. We can invoke a different solver using the flag `-b`. To invoke the mixed integer programming solver (`mip`) which can handle (linear) real number solving we invoke the command

```
$ mzn -b mip loan.mzn loan1.dzn
```

The output is

```
Borrowing 1000.0 at 4.0% interest, and repaying 260.0
per quarter for 1 year leaves 65.77792000000005 owing
----------
```

The second question is if I want to borrow $1000 at 4% and owe nothing at the end, how much do I need to repay? This question is encoded by the data file `loan2.dzn`. The output from running the command

```
$ mzn -b mip loan.mzn loan2.dzn
```

is

```
Borrowing 1000.0 at 4.0% interest, and repaying 275.49004536480237
per quarter for 1 year leaves 0.0 owing
----------
```

The third question is if I can repay $250 a quarter, how much can I borrow at 4% to end up owing nothing? This question is encoded by the data file `loan3.dzn`. The output from running the command

loan1.dzn

```
I = 0.04;
P = 1000.0;
R = 260.0;
```

loan2.dzn

```
I = 0.04;
P = 1000.0;
B4 = 0.0;
```

loan3.dzn

```
I = 0.04;
R = 250.0;
B4 = 0.0;
```

Figure 7: Example data files for `loan.mzn`

```
$ mzn -b mip loan.mzn loan3.dzn
is  Borrowing 907.4738060642132 at 4.0% interest, and repaying 250.0
per quarter for 1 year leaves 0.0 owing
----------
```

Floating point arithmetic Operators:
MiniZinc provides the standard floating point arithmetic operators. Addition (`+`), subtraction (`-`), multiplication (`*`) and floating point division (`/`). It also provides `+` and `-` as unary operators.
MiniZinc does not automatically coerce integers to floating point numbers. The built-in function `int2float` can be used for this purpose.
MiniZinc also includes the floating point functions for absolute value (`abs`), square root (`sqrt`), natural logarithm (`ln`), logarithm base 2 (`log2`), logarithm base 10 (`log10`), exponentiation of e (`exp`), sine (`sin`), cosine (`cos`), tangent (`tan`), arcsine (`asin`), arccosine (`acos`), arctangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), hyperbolic arcsine (`asinh`), hyperbolic arccosine (`acosh`), hyperbolic arctangent (`atanh`), and power (`pow`) which is the only binary function, the rest are unary.
The syntax for arithmetic literals is reasonably standard. Example float literals are `1.05`, `1.3e-5` and `1.3+E5`.

2.5 Basic structure of a model

We are now in a position to summarise the basic structure of a MiniZinc model. It consists of multiple items each of which has a semicolon '`;`' at its end. Items can occur in any order. For example, identifiers need not be declared before they are used.

There are 7 kinds of items.

- Include items allow the contents of another file to be inserted into the model. They have form

  `include` ⟨*filename*⟩;

  where *filename* is a string literal. They allow large models to be split into smaller sub-models and also the inclusion of constraints defined in library files. We shall see an example in Figure 11.

- Variable declarations declare new variables. Such variables are global variables and can be referred to from anywhere in the model. Variables come in two kinds. Parameters which are assigned a fixed value in the model or in a data file and decision variables whose value is found only when the model is solved. We say that parameters are *fixed* and decision variables *unfixed*. The variable can be optionally assigned a value as part of the declaration. The form is:

⟨*type inst expr*⟩: ⟨*variable*⟩ [= ⟨*expression*⟩];

The *type inst expr* gives the instantiation and type of the variable. These are one of the more complex aspects of MiniZinc . Instantiations are declared using `par` for parameters and `var` for decision variables. If there is no explicit instantiation declaration then the variable is a parameter. The type can be a base type, an integer or float range or an array or a set. The base types are `float`, `int`, `string`, `bool`, `ann` of which only `float`, `int` and `bool` can be used for decision variables. The base type `ann` is an annotation—we shall discuss annotations in Section 5. Integer range expressions can be used instead of the type `int`. Similarly float range expressions can be used instead of type `float`. These are typically used to give the domain of an integer decision variable but can also be used to restrict the range of an integer parameter.

- Assignment items assign a value to a variable. They have form:

  ⟨*variable*⟩ = ⟨*expression*⟩;

  Values can be assigned to decision variables in which case the assignment is equivalent to writing `constraint` ⟨*variable*⟩ = ⟨*expression*⟩;

- Constraint items form the heart of the model. They are of form

  `constraint` ⟨*Boolean expression*⟩;

  We have already seen examples of simple constraints using arithmetic comparison and the built-in *assert* operator. In the next section we shall see examples of more complex constraints.

- Solve items specify exactly what kind of solution is being looked for. As we have seen they have one of three forms

  ```
  solve satisfy;
  solve maximize ⟨arithmetic expression⟩;
  solve minimize ⟨arithmetic expression⟩;
  ```

  A model is required to have exactly one solve item.

- Output items are for nicely presenting the results of the model execution. They have form

  `output [` ⟨*string expression*⟩`,` ⋯ `,` ⟨*string expression*⟩ `];`

- Predicate and test items are for defining new constraints and Boolean tests. We discuss these in Section 4.

- The annotation item is used to define a new annotation. We discuss these in Section 5.

3 More Complex Models

In the last section we introduced the basic structure of a MiniZinc model. In this section we introduce the array and set data structures and more complex constraints.

3.1 Arrays and Sets

Almost always we are interested in building models where the number of constraints and variables is dependent on the input data. In order to do so we will usually use arrays.

Consider a simple finite element model for modelling temperatures on a rectangular sheet of metal. We approximate the temperatures across the sheet by breaking the sheet into a finite number of elements in a 2 dimensional matrix. A model is shown in Figure 8. It declares the width `w` and height `h` of the finite element model. The declaration

```
array[0..w, 0..h] of var float: t;
```

declares a two dimensional array of float variables `t` with rows numbered 0 to w and columns 0 to h, to represent the temperatures at each point in the metal plate. We can access the element of the array in the i^{th} row and j^{th} column using an expression `t[` i `,` j `]`.

Laplace's equation states that when the plate reaches a steady state the temperature at each internal point is the average of its orthogonal neighbours. The first constraint ensures each internal point (i, j) to the average of its four orthogonal neighbours. The model then constrains the temperatures on each edge to be equal, and gives these temperatures names: `left`, `right`, `top` and `bottom`. We can determine the temperatures in a plate broken into 5×5 elements with left, right and bottom temperature 0 and top temperature 100 with the model shown in Figure 8.

Running the command

```
$ mzn -b mip laplace.mzn
```

gives the output

```
0.0 100.0 100.0 100.0 0.0
0.0 42.857142857142854 52.67857142857142 42.857142857142854 0.0
0.0 18.750000000000004 24.999999999999996 18.75 0.0
0.0 7.142857142857143 9.82142857142857 7.142857142857142 0.0
0.0 0.0 0.0 0.0 0.0
----------
```

Our cake baking problem is an example of a very simple kind of production planning problem. In this kind of problem we wish to determine how much of each kind of product to make to maximise the profit where manufacturing a product consumes varying amounts of some fixed resources. We can generalise the MiniZinc model in Figure 4 to handle this kind of problem with a model that is generic in the kinds of resources and products. The model is shown in Figure 9 and a sample data file (for the cake baking example) is shown in Figure 10.

The new feature in the model is the use of arrays and sets. These allow us to treat the choice of resources and products as parameters to the model. The first item in the model

```
int:  nproducts;
```

declares that the number of products `nproducts` is a parameter. The next item

```
set of int:  Products = 1..nproducts;
```

laplace.mzn

```
int: w = 4;
int: h = 4;

array[0..w,0..h] of var float: t; % temperature at point (i,j)
var float: left;   % left   edge temperature
var float: right;  % right  edge temperature
var float: top;    % top    edge temperature
var float: bottom; % bottom edge temperature

% Laplace equation: each internal temperature is average of its neighbours
constraint forall(i in 1..w-1, j in 1..h-1)(
              4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);

% edge constraints
constraint forall(i in 1..w-1)(t[i,0] = left);
constraint forall(i in 1..w-1)(t[i,h] = right);
constraint forall(j in 1..h-1)(t[0,j] = top);
constraint forall(j in 1..h-1)(t[w,j] = bottom);

% corner constraints
constraint t[0,0] = 0.0 /\ t[0,h] = 0.0 /\ t[w,0] = 0.0 /\ t[w,h] = 0.0;

left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output   [ show(t[i,j]) ++ if j == h then "\n" else " " endif |
         i in 0..w, j in 0..h ];
```

Figure 8: Finite element plate model for determining steady state temperatures (`laplace.mzn`).

declares a new parameter `Products`, which is the set of products. It is initialized to the set $\{1, 2, ..., nproducts\}$ using a range expression.

The third item declares an array of integers:

```
array[Products] of int:  profit;
```

The index set of the array `profit` is `products`. This means $1, 2, ..., nproducts$ are valid indices into the array. The array access `profit[i]` gives the profit for product `i`. The following are equivalent declarations:

```
array[1..nproducts] of int:  profit;
array[Products] of par int:  profit;
```

In the example data file we have initialized the array using a list of integers

```
profit = [400,450];
```

While MiniZinc does not provide an explicit list type, one-dimensional arrays with an index set $1..n$ behave like lists, and we will sometimes refer to them as lists. The fourth item define an array of strings, the product names:

```
array[Products] of string:  name;
```

In a similar fashion, in the next 4 items we declare a parameter `nresources` which is the number of resources, a set of resources `resources`, an array `capacity`

simple-prod-planning.mzn

```
% Number of different products
int: nproducts;
set of int: Products = 1..nproducts;

%profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;

%Number of resources
int: nresources;
set of int: Resources = 1..nresources;

%amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;

%units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
           (consumption[p,r] >= 0), "Error: negative consumption");

% bound on number of Products
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                          (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
      used[r] = sum (p in Products)(consumption[p, r] * produce[p])
      /\ used[r] <= capacity[r]
);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n" |
         p in Products ] ++
       [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" |
         r in Resources ];
```

Figure 9: Model for a simple production planning problem (`simple-prod-planning.mzn`).

which gives the amount of each resource that is available, and an array `rname` of resource names.

More interestingly, the item

`array[Products, Resources] of int: consumption;`

declares a 2-D array `consumption`. The value of `consumption[p,r]` is the amount of resource `r` required to produce one unit of product `p`. Note that the first index is the row and the second is the column.

simple-prod-planning.dzn

```
% Data file for simple production planning model

nproducts = 2;       %banana cakes and chocolate cakes
profit = [400, 450]; %in cents
pname = ["banana-cake", "chocolate-cake"];

nresources = 5;      %flour, banana, sugar, butter, cocoa
capacity = [4000, 6, 2000, 500, 500];
rname = ["flour","banana","sugar","butter","cocoa"];

consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

Figure 10: Example data file for the simple production planning problem model shown in Figure 9.

Sets:
Set variables are declared with a declaration of form

`set of` ⟨*type-inst*⟩

where sets of integers, floats or Booleans are allowed. Set literals are of form

{ ⟨$expr_1$⟩, ... , ⟨$expr_n$⟩ }

or are range expressions over either integers or floats of form

⟨$expr_1$⟩ `..` ⟨$expr_2$⟩

The standard set operations are provided: element membership (`in`), set containment (`subset`, `superset`), union (`union`), intersection (`inter`), set difference (`diff`), symmetric set difference (`symdiff`) and the number of elements in the set (`card`).
As we have seen set variables and set literals (including ranges) can be used as an implicit type in variable declarations in which case the variable has the type of the elements in the set and the variable is implicitly constrained to be a member of the set.

The data file contains an example initialization of a 2-D array:
`consumption= [| 250, 2, 75, 100, 0,`
`| 200, 0, 150, 150, 75 |];`

Notice how the delimiter `|` is used to separate rows. The next item in the model defines the parameter `mproducts`. This is set to an upper-bound on the number of products of any type that can be produced. This is quite a complex example of nested array comprehensions and aggregation operators. We shall introduce these before we try and understand this item and the rest of the model.

First, MiniZinc provides list comprehensions similar to those provided in many functional programming languages. For example, the list comprehension `[i + j | i, j in 1..3 where j < i]` evaluates to `[1 + 2, 1 + 3, 2 + 3]` which is `[3, 4, 5]`. Of course `[3, 4, 5]` is simply an array with index set 1..3.

MiniZinc also provides set comprehensions which have a similar syntax: for instance, {`i + j | i, j in 1..3 where j < i`} evaluates to the set {`3, 4, 5`}.

Arrays:
Thus, MiniZinc provides one- and multi-dimensional arrays which are declared using the type:

`array[` ⟨*index-set*$_1$⟩, ..., ⟨*index-set*$_n$⟩ `]` `of` ⟨*type-inst*⟩

MiniZinc requires that the array declaration contains the index set of each dimension and that the index set is either an integer range or a set variable initialised to an integer range. Arrays can contain any of the base types: integers, booleans, floats or strings. These can be fixed or unfixed except for strings which can only be parameters. Arrays can also contain sets but they cannot contain arrays.
One-dimensional array literals are of form

`[` ⟨*expr*$_1$⟩, ... , ⟨*expr*$_n$⟩ `]`

while two-dimensional array literals are of form

`[|` ⟨*expr*$_{1,1}$⟩, ... , ⟨*expr*$_{1,n}$⟩, `|` ..., `|` ⟨*expr*$_{m,1}$⟩, ... , ⟨*expr*$_{m,n}$⟩ `|]`

where the array has m rows and n columns.
The family of built-in functions `array1d`, `array2d`, etc, can be used to initialise an array of any dimension from a list (or more exactly a one-dimensional array). The call:

`array`*n*`d`(⟨*index-set*$_1$⟩, ..., ⟨*index-set*$_n$⟩, ⟨*list*⟩ `)`

returns an n dimensional array with index sets given by the first n arguments and the last argument contains the elements of the array. For instance, `array2d(1..3, 1..2, [1, 2, 3, 4, 5, 6])` is equivalent to `[|1, 2 |3, 4 |5, 6|])`.
Array elements are accessed in the usual way: `a[i,j]` gives the element in the i^{th} row and j^{th} column.
The concatenation operator '`++`' can be used to concatenate two one-dimensional arrays together. The result is a list, i.e. a one-dimensional array whose elements are indexed from 1. For instance `[4000, 6] ++ [2000, 500, 500]` evaluates to `[4000, 6, 2000, 500, 500]`. The built-in function `length` returns the length of a one-dimensional array.

Second, MiniZinc provides a number of built-in functions that take a one-dimensional array and aggregate the elements. Probably the most useful of these is `forall`. This takes an array of Boolean expressions (that is, constraints) and returns a single Boolean expression which is the logical conjunction of the Boolean expressions in the array.

For example, consider the expression

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

where `a` is an arithmetic array with index set 1..3. This constrains the elements in `a` to be different. The list comprehension evaluates to `[ a[1] != a[2], a[1] != a[3], a[2] != a[3] ]` and so the `forall` function returns the logical conjunction `a[1] != a[2]` ∧ `a[1] != a[3]` ∧ `a[2] != a[3]`.

The third, and final, piece in the puzzle is that MiniZinc allows a special syntax for aggregation functions when used with an array comprehension. Instead of writing

List and Set Comprehensions:
The generic form of a list comprehension is

`[` ⟨*expr*⟩ `|` ⟨*generator-exp*⟩ `]`

The expression ⟨*expr*⟩ specifies how to construct elements in the output list from the elements generated by ⟨*generator-exp*⟩. The generator ⟨*generator-exp*⟩ consists of a comma separated sequence of generator expressions optionally followed by a Boolean expression. The two forms are

⟨*generator*⟩, ..., ⟨*generator*⟩
⟨*generator*⟩, ..., ⟨*generator*⟩ `where` ⟨*bool-exp*⟩

The optional ⟨*bool-exp*⟩ in the second form acts as a filter on the generator expression: only elements satisfying the Boolean expression are used to construct elements in the output list. A generator ⟨*generator*⟩ has form

⟨*identifier*⟩,..., ⟨*identifier*⟩ `in` ⟨*array-exp*⟩

Set comprehensions are almost identical to list comprehensions: the only difference is the use of '{' and '}' to enclose the expression rather than '[' and ']'. The elements generated by a set comprehension have to be fixed, i.e. free of decision variables.

Aggregation functions:
The *aggregation functions* for arithmetic arrays are: `sum` which adds the elements, `product` which multiplies them together, and `min` and `max` which respectively return the least and greatest element in the array. When applied to an empty array, `min` and `max` give a run-time error, `sum` returns 0 and `product` returns 1.
MiniZinc provides two aggregation functions for arrays containing Boolean expressions. As we have seen, the first of these, `forall`, returns a single constraint which is the logical conjunction of the constraints. The second function, `exists`, returns the logical disjunction of the constraints. Thus, `forall` enforces that all constraints in the array hold, while `exists` ensures that at least one of the constraints holds.

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

the modeller can instead write the more mathematical looking

```
forall (i,j in 1..3 where i < j) (a[i] != a[j])
```

The two expressions are completely equivalent: the modeller is free to use whichever they feel looks most natural.

We are now in a position to understand the rest of the simple production planning model shown in Figure 9. For the moment ignore the item defining `mproducts`. The item afterwards:

```
array[Products] of var 0..mproducts: produce;
```

defines a one-dimensional array `produce` of decision variables. The value of `produce[p]` will be set to the amount of product `p` in the optimal solution. The next item

Generator call expressions:
A *generator call expression* has form

⟨*agg-func*⟩ (⟨*generator-exp*⟩) (⟨*exp*⟩)

The round brackets around the generator expression ⟨*generator-exp*⟩ and the constructor expression ⟨*exp*⟩ are not optional: they must be there. This is equivalent to writing

⟨*agg-func*⟩([⟨*expr*⟩ | ⟨*generator-exp*⟩])

The aggregation function ⟨*agg-func*⟩ is any of the built-in MiniZinc aggregation functions.

```
array[Resources] of var 0..max(capacity): used;
```

defines a set of auxiliary variables that record how much of each resource is used. The next item

```
constraint forall (r in Resources) (
             used[r] = sum (p in products) (consumption[p, r] * produce[p])
          /\ used[r] <= capacity[r] );
```

computes in `used[r]` the total consumption of the resource `r` and ensures it is less than the available amount. Finally, the item

```
solve maximize sum (p in Products) (profit[p]*produce[p]);
```

indicates that this is a maximisation problem and that the objective to be maximised is the total profit.

We now return to the definition of `mproducts`. For each product `p` the expression

```
(min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r])
```

determines the maximum amount of `p` that can be produced taking into account the amount of each resource `r` and how much of `r` is required to produce the product. Notice the use of the filter `where consumption[p,r] > 0` to ensure that only resources required to make the product are considered so as to avoid a division by zero error. Thus, the complete expression

```
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r]));
```

computes the maximum amount of *any* product that can be produced, and so this can be used as an upper bound on the domain of the decision variables in `produce`.

Finally notice the output item is more complex, and uses list comprehensions to create an understandable output. Running

```
$ mzn simple-prod-planning.mzn simple-prod-planning.dzn
```

results in the output

```
banana-cake = 2;
chocolate-cake = 2;
flour = 900;
```

send-more-money.mzn

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint           1000 * S + 100 * E + 10 * N + D
                   + 1000 * M + 100 * O + 10 * R + E
       = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output ["   ",show(S),show(E),show(N),show(D),"\n",
        "+  ",show(M),show(O),show(R),show(E),"\n",
        "= ",show(M),show(O),show(N),show(E),show(Y),"\n"];
```

Figure 11: Mode for the cryptarithmetic problem SEND+MORE=MONEY (`send-more-money.mzn`).

```
banana = 4;
sugar = 450;
butter = 500;
cocoa = 150;
----------
```

3.2 Global Constraints

MiniZinc includes a library of global constraints which can also be used to define models. An example is the `alldifferent` constraint which requires all the variables appearing in its argument to be different.

The SEND+MORE=MONEY problem requires assigning a different digit to each letter so that the arithmetic constraint holds. The model shown in Figure 11 uses the constraint expression `alldifferent([S,E,N,D,M,O,R,Y])` to ensure that each letter takes a different digit value. The global constraint is made available in the model using include item

```
include "alldifferent.mzn";
```

which makes the global constraint `alldifferent` usable by the model. One could replace this line by

```
include "globals.mzn";
```

which includes all globals.

A list of all the global constraints defined for MiniZinc is included in the release documentation. See Section 4.1 for a description of some important global constraints.

3.3 Conditional Expressions

MiniZinc provides a conditional *if-then-else-endif* expression. An example of its use is

```
int: r = if y != 0 then x div y else 0 endif;
```

which sets r to x divided by y unless y is zero in which case it sets it to zero.

> Conditional expressions:
> The form of a conditional expression is
>
> `if` ⟨*boolexp*⟩ `then` ⟨exp_1⟩ `else` ⟨exp_2⟩ `endif`
>
> It is a true expression rather than a control flow statement and so can be used in other expressions. It evaluates to ⟨exp_1⟩ if the Boolean expression ⟨*boolexp*⟩ is true and ⟨exp_2⟩ otherwise. The type of the conditional expression is that of ⟨exp_1⟩ and ⟨exp_2⟩ which must have the same type.
> The Boolean expression is not allowed to contain decision variables, only parameters.
> In output items the built-in function `fix` checks that the value of a decision variable is fixed and coerces the instantiation from decision variable to parameter.

Conditional expressions are very useful in building complex models, or complex output. Consider the model of Sudoku problems shown in Figure 12. The initial board positions are given by the `start` parameter where 0 represents an empty board position. This is converted to constraints on the decision variables `puzzle` using the conditional expression

```
constraint forall(i,j in PuzzleRange)(
     if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );
```

Conditional expressions are also very useful for defining complex output. In the Sudoku model of Figure 12 the expression

```
if fix(puzzle[i,j]) <= 9  then " " else "" endif ++ show(puzzle[i,j])
```

outputs the value of the square as two characters. The output expression also use conditional expressions to insert an extra space in between groups of size S, and add blank lines after each S lines. The resulting output is highly readable.

The remaining constraints ensure that the numbers appearing in each row and column and $S \times S$ subsquare are all different.

One can use MiniZinc to search for all solutions to a satisfaction problem (`solve satisfy`) by using the flag `-a` or `--all-solutions`.

```
$ mzn --all-solutions sudoku.mzn sudoku.dzn
```

results in

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

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

sudoku.mzn

```
include "alldifferent.mzn";

int: S;
int: N = S * S;

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
        if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint
        forall (a, o in SubSquareRange)(
                alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                                    a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ if fix(puzzle[i,j]) <= 9  then " " else "" endif
          ++ show(puzzle[i,j]) ++ " " ++
         if j mod S == 0 then " " else "" endif ++
         if j == N /\ i != N then
             if i mod S == 0 then "\n\n" else "\n" endif
         else "" endif
         | i,j in PuzzleRange ] ++ ["\n"];
```

Figure 12: Model for generalized Sudoku problem (`sudoku.mzn`).

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

9 4 2  6 1 8  7 5 3
8 3 5  2 4 7  6 9 1
6 7 1  5 9 3  4 8 2
----------
==========
```

Here the output verifies there is only one solution. The line `==========` is output when the system has output all possible solutions, here verifying that there is exactly one.

sudoku.dzn

```
S=3;
start=[|
0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 6, 8, 4, 0, 1, 0, 7, 0|
0, 0, 0, 0, 8, 5, 0, 3, 0|
0, 2, 6, 8, 0, 9, 0, 4, 0|
0, 0, 7, 0, 0, 0, 9, 0, 0|
0, 5, 0, 1, 0, 6, 3, 2, 0|
0, 4, 0, 6, 1, 0, 0, 0, 0|
0, 3, 0, 2, 0, 7, 6, 9, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0|];
```

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

Figure 13: Example data file for generalised Sudoku problem (`sudoku.dzn`) and the problem it represents.

3.4 Complex Constraints

Constraints are the core of the MiniZinc model. We have seen simple relational expressions but constraints can be considerably more powerful than this. A constraint is allowed to be any Boolean expression. Imagine a scheduling problem in which we have two tasks that cannot overlap in time. If `s1` and `s2` are the corresponding start times and `d1` and `d2` are the corresponding durations we can express this as:

```
constraint s1 + d1 <= s2  \/ s2 + d2 <= s1;
```

which ensures that the tasks do not overlap.

> Booleans:
> Boolean expressions in MiniZinc can be written using a standard mathematical syntax. The Boolean literals are `true` and `false` and the Boolean operators are conjunction, i.e and (`/\`), disjunction, i.e or (`\/`), only-if (`<-`), implies (`->`), if-and-only-if (`<->`) and negation (`not`). The built-in function `bool2int` coerces Booleans to integers: it returns 1 if its argument is true and 0 otherwise.

The job shop scheduling model given in Figure 14 gives a realistic example of the use of this disjunctive modelling capability. In job shop scheduling we have a set of jobs, each consisting of a sequence of tasks on separate machines: so task $[i, j]$ is the task in the i^{th} job performed on the j^{th} machine. Each sequence of tasks must be completed in order, and no two tasks on the same machine can overlap in time. Even small instances of this problem can be quite challenging to find optimal solutions.

The command

```
$ mzn --all-solutions jobshop.mzn jobshop.dzn
```

solves a small job shop scheduling problem, and illustrates the behaviour of `all-solutions` for optimisation problems. Here the solver outputs each better solutions as it finds it, rather than all possible optimal solutions. The (partial) output from this command is:

```
end = 41
0 1 5 10 13
```

jobshop.mzn

```
int: jobs;                                      % no of jobs
int: tasks;                                     % no of tasks per job
array [1..jobs,1..tasks] of int: d;             % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
             (d[i,j]);                          % total duration
array [1..jobs,1..tasks] of var 0..total: s;    % start times
var 0..total: end;                              % total end time


constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        forall(i,k in 1..jobs where i < k) (
            s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show(s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```

Figure 14: Model for job-shop scheduling problems (`jobshop.mzn`).

```
5 8 10 25 26
1 10 17 21 28
8 14 21 26 32
9 16 22 32 40
----------

    ...

----------
end = 31
0 3 7 12 18
6 9 19 26 28
2 11 15 19 24
1 2 3 4 10
9 16 26 28 30
----------
end = 30
1 2 6 11 17
6 10 15 22 23
```

stable-marriage.mzn

```
int: n;

set of int: Men = 1..n;
set of int: Women = 1..n;

array[Women, Men] of int: rankWomen;
array[Men, Women] of int: rankMen;

array[Men] of var Women: wife;
array[Women] of var Men: husband;

constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);

constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
         rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
         rankMen[o,wife[o]] < rankMen[o,w] );

solve satisfy;

output ["wives= ", show(wife),"\n", "husbands= ", show(husband), "\n"];
```

Figure 15: Model for stable marriage problem (`stable-marriage.mzn`).

```
2 6 11 15 25
0 1 2 3 9
9 16 22 24 29
----------
==========
```

indicating an optimal solution with end time 30 is finally found, and proved optimal. We can generate all *optimal solutions* by adding a constraint that `end = 30` and changing the solve item to `solve satisfy` and then executing

```
$ mzn --all-solutions jobshop.mzn jobshop.dzn
```

For this problem there are very many optimal solutions.

Another powerful modelling feature in MiniZinc is that decision variables can be used for array access. As an example, consider the (old-fashioned) *stable marriage problem*. We have n (straight) women and n (straight) men. Each man has a ranked list of women and vice versa. We want to find a husband/wife for each women/man so that all marriages are *stable* in the sense that:

- whenever m prefers another women o to his wife w, o prefers her husband to m, and
- whenever w prefers another man o to her husband m, o prefers his wife to w.

This can be elegantly modelled in MiniZinc . The model and sample data is shown in Figures 15 and 16. The model is based on a model written in the OPL modelling language.

stable-marriage.dzn

```
n = 5;
rankWomen =
 [| 1, 2, 4, 3, 5,
  | 3, 5, 1, 2, 4,
  | 5, 4, 2, 1, 3,
  | 1, 3, 5, 4, 2,
  | 4, 2, 3, 5, 1 |];

rankMen =
 [| 5, 1, 2, 4, 3,
  | 4, 1, 3, 2, 5,
  | 5, 3, 2, 4, 1,
  | 1, 5, 4, 3, 2,
  | 4, 3, 2, 1, 5 |];
```

Figure 16: Example data file for the stable marriage problem model shown in Figure 15.

The first three items in the model declare the number of men/women and the set of men and women. The matrices `rankWomen` and `rankMen`, respectively, give the women's ranking of the men and the men's ranking of the women. Thus, the entry `rankWomen[w,m]` gives the ranking by woman `w` of man `m`. The lower the number in the ranking, the more the man or women is preferred.

There are two arrays of decision variables: `wife` and `husband`. These, respectively, contain the wife of each man and the husband of each women.

The first two constraints

```
constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);
```

ensure that the assignment of husbands and wives is consistent: w is the wife of m implies m is the husband of w and vice versa. Notice how in `husband[wife[m]]` the index expression `wife[m]` is a decision variable, not a parameter.

The next two constraints are a direct encoding of the stability condition:

```
constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
         rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
         rankMen[o,wife[o]] < rankMen[o,w] );
```

This natural modelling of the stable marriage problem is made possible by the ability to use decision variables as array indices and to construct constraints using the standard Boolean connectives. The alert reader may be wondering at this stage, what happens if the array index variable takes a value that is outside the index set of the array. MiniZinc treats this as failure: an array access `a[e]` implicitly adds the constraint `e in index_set(a)` to the closest surrounding Boolean context where `index_set(a)` gives the index set of `a`.

Thus for example, consider the variable declarations

```
array[1..2] of int: a= [2,3];
var 0..2: x;
var 2..3: y;
```

magic-series.mzn

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

Figure 17: Model solving the magic series problem (`magic-series.mzn`).

The constraint `a[x] = y` will succeed with $x = 1 \wedge y = 2$ and $x = 2 \wedge y = 3$. And the constraint `not a[x] = y` will succeed with $x = 0 \wedge y = 2$, $x = 0 \wedge y = 3$, $x = 1 \wedge y = 3$ and $x = 2 \wedge y = 2$.

In the case of invalid array accesses by a parameter, the formal semantics of MiniZinc treats this as failure so as to ensure that the treatment of parameters and decision variables is consistent, but a warning is issued since it is almost always an error.

The coercion function `bool2int` can be called with any Boolean expression. This allows the MiniZinc modeller to use so called *higher order constraints*. As an simple example consider the *magic series problem*: find a list of numbers $s = [s_0, \ldots, s_{n-1}]$ such that s_i is the number of occurrences of i in s. An example is $s = [1, 2, 1, 0]$.

A MiniZinc model for this problem (based on an OPL model given in []) is shown in Figure 17. The use of `bool2int` allows us to sum up the number of times the constraint `s[j]=i` is satisfied. Executing the command

```
$ mzn --all-solutions magic-series.mzn -D "n=4;"
```

leads to the output

```
s = [1, 2, 1, 0];
----------
s = [2, 0, 2, 0];
----------
==========
```

indicating exactly two solutions to the problem.

3.5 Set Constraints

Another powerful modelling feature of MiniZinc is that it allows sets containing integers to be decision variables: this means that when the model is evaluated the solver will find which elements are in the set.

As a simple example, consider the *0/1 knapsack problem*. This is a restricted form of the knapsack problem in which we can either choose to place the item in the knapsack or not. Each item has a weight and a profit and we want to find which choice of items leads to the maximum profit subject to the knapsack not being too full.

It is natural to model this in MiniZinc with a single decision variable:

```
var set of items: knapsack;
```

knapsack.mzn

```
int: n;
set of int: Items = 1..n;
int: capacity;

array[Items] of int: profits;
array[Items] of int: weights;

var set of Items: knapsack;

constraint sum (i in Items)
               (bool2int(i in knapsack)*weights[i]) <= capacity;

solve maximize sum (i in Items) (bool2int(i in knapsack)*profits[i]) ;

output [show(knapsack),"\n"];
```

Figure 18: Model for the 0/1 knapsack problem (`knapsack.mzn`).

where `items` is the set of possible items. If the arrays `weight[i]` and `profit[i]` respectively give the weight and profit of item `i`, and the maximum weight the knapsack can carry is given by `capacity` then it is natural to write the capacity constraint as

```
constraint sum (i in knapsack) (weights[i]) <= capacity;
```

and the solve item as

```
solve maximize sum (i in knapsack) (profits[i]) ;
```

Unfortunately, MiniZinc does not allow this and the capacity constraint and solve item will lead to a type-inst error stating that a generator expression must iterate over an array (or a fixed set, which can be coerced into an array) and cannot iterate over a `var` set.

At first glance, this seems an insurmountable problem. However, we can use the coercion function `bool2int` to overcome the problem. For the capacity constraint we write

```
constraint sum (i in Items) (bool2int(i in knapsack)*weights[i])
           <= capacity;
```

Our generator expression now iterates over the domain of `knapsack` which is fixed and uses `bool2int` to appropriately transform the weight depending on whether the item is in `knapsack` or not. We can use a similar trick to rewrite the solve item. This leads to the MiniZinc model shown in Figure 18 which is a valid MiniZinc model.

Notice that the `var` keyword comes before the `set` declaration indicating that the set itself is the decision variable. This contrasts with an array in which the `var` keyword qualifies the elements in the array rather than the array itself since the basic structure of the array is fixed, i.e. its index set.

As a more complex example of set constraint consider the social golfers problem shown in Figure 19. The aim is to schedule a golf tournament over `weeks` using `groups` × `size` golfers. Each week we have to schedule `groups` different groups each of size `size`. No two pairs of golfers should ever play in two groups.

The variables in the model are sets of golfers `Sched[i,j]` for the i^{th} week and j^{th} group, The constraint first enforces an ordering on the first set in each week to

social-golfers.mzn

```
include "partition_set.mzn";
int: weeks;
int: groups;
int: size;
int: ngolfers = groups*size;

array[1..weeks,1..groups] of var set of 1..ngolfers: Sched;

constraint
    forall (i in 1..weeks-1) (
                Sched[i,1] < Sched[i+1,1]
    ) /\
    forall (i in 1..weeks, j in 1..groups) (
           card(Sched[i,j]) = size
        /\ forall (k in j+1..groups) (
                 Sched[i,j] < Sched[i,k]
              /\ Sched[i,j] intersect Sched[i,k] = {}
           )
    ) /\
    forall (i in 1..weeks) (
          partition_set([Sched[i,j] | j in 1..groups], 1..ngolfers)
       /\ forall (j in 1..groups-1) (
              Sched[i,j] < Sched[i,j+1]
          )
    ) /\
    forall (i in 1..weeks-1, j in i+1..weeks) (
        forall (x in 1..groups, y in 1..groups) (
            card(Sched[i,x] intersect Sched[j,y]) <= 1
        )
    );

constraint
    % Fix the first week %
    forall (i in 1..groups, j in 1..size) (
        ((i-1)*size + j) in Sched[1,i]
    ) /\
    % Fix first group of second week %
    forall (i in 1..size) (
        ((i-1)*size + 1) in Sched[2,1]
    ) /\
    % Fix first 'size' players
    forall (w in 2..weeks, p in 1..size) (
        p in Sched[w,p]
    );

solve :: set_search([Sched[i,1] | i in 1..weeks] ++
                        [Sched[i,j] | i in 1..weeks, j in 2..groups],
                        input_order, indomain_min, complete) satisfy;

output [ show(Sched[i,j]) ++ " " ++
         if j == groups then "\n" else "" endif |
         i in 1..weeks, j in 1..groups ];
```

Figure 19: Model for the social golfers problems (`social-golfers.mzn`).

remove symmetry in swapping weeks. Next it enforces an ordering on the sets in each week, and makes each set have a cardinality of `size`. It then ensures that each week is a partition of the set of golfers using the global constraint `partition_set`. Finally the last constraint ensures that no two players play in two groups together (since the cardinality of the intersection of any two groups is at most 1).

There are also symmetry breaking initialisation constraints: the first week is fixed to have all players in order; the second week is made up of the first players of each of the first groups in the first week; finally the model forces the first `size` players to appear in their corresponding group number for the remaining weeks.

Executing the command

```
$ mzn social-golfers.mzn social-golfers.dzn
```

where the data file defines a problem with 4 weeks, with 4 groups of size 3 leads to the output

```
1..3 4..6 7..9 10..12
{1, 4, 7} {2, 5, 10} {3, 9, 11} {6, 8, 12}
{1, 5, 8} {2, 6, 11} {3, 7, 12} {4, 9, 10}
{1, 6, 9} {2, 4, 12} {3, 8, 10} {5, 7, 11}
----------
```

Notice hows sets which are ranges may be output in range format.

3.6 Enumerated Types

MiniZinc does not include enumerated types, but they can be mimicked using a style of modelling. The enumerated type is actually a range of integers $1..n$ and each element in the type is a parameter with different value in the range.

The model of Figure 20 arranges seats at the wedding table. The table has 12 numbered seats in order around the table, 6 on each side. Males must sit in odd numbered seats, and females in even. Ed cannot sit at the end of the table because of a phobia, and the bride and groom must sit next to each other. The aim is to maximize the distance between known hatreds. The distance between seats is the difference in seat number if on the same side, otherwise its the distance to the opposite seat + 1. The enumerated type `Guests` is implemented by a range, and using integer parameters to name each guest. The rest of the model uses the guest names.

Running

```
$ mzn wedding.mzn
```

Results in the output

```
ted bride groom rona bob carol ron alice ed bridesmaid bestman clara
----------
==========
```

The resulting table placement is illustrated in Figure 21 where the lines indicate hatreds. The total distance is 22.

wedding.mzn

```
set of int: Guests = 1..12;
int: bride = 1; int: groom = 2; int: bestman = 3;
int: bridesmaid = 4; int: bob = 5; int: carol  = 6;
int: ted = 7; int: alice = 8; int: ron = 9;
int: rona = 10; int: ed = 11; int: clara = 12;
array[Guests] of string: name = ["bride","groom","bestman",
     "bridesmaid","bob","carol","ted","alice","ron","rona","ed","clara"];
set of int: Seats = 1..12;

set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];

set of int: Males = {groom, bestman, bob, ted, ron,ed};
set of int: Females = {bride, bridesmaid, carol, alice, rona, clara};

array[Guests] of var Seats: pos; % seat of guest
array[Hatreds] of var Seats: p1;  % seat of guest 1 in hatred
array[Hatreds] of var Seats: p2;  % seat of guest 2 in hatred
array[Hatreds] of var 0..1: sameside;  % seats of hatred on same side
array[Hatreds] of var Seats: cost;  % penalty of hatred

include "alldifferent.mzn";

constraint alldifferent(pos);

constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );

constraint pos[ed] != 1 /\ pos[ed] != 6 /\ pos[ed] != 7 /\ pos[ed] != 12;
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);

constraint forall(h in Hatreds)(
               p1[h] = pos[h1[h]] /\
               p2[h] = pos[h2[h]] /\
               sameside[h] = bool2int(p1[h] <= 6 <-> p2[h] <= 6) /\
               cost[h] = sameside[h] * abs(p1[h] - p2[h]) +
                     (1 - sameside[h]) * (abs(13 - p1[h] - p2[h]) + 1)
           );

solve maximize sum(h in Hatreds)(cost[h]);

output [ name[g] ++ " " | s in Seats, g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

Figure 20: Planning wedding seating using enumerated types (`wedding.mzn`)

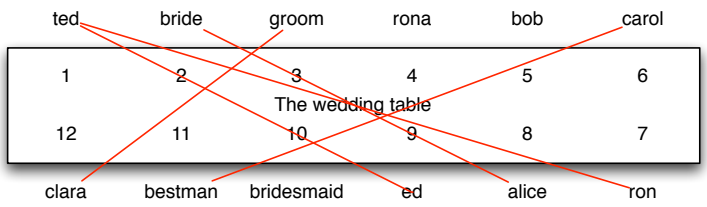

Figure 21: Seating arrangement at the wedding table

4 Predicates

Predicates in MiniZinc allow us to capture complex constraints of our model in a succinct way. Predicates in MiniZinc are used to model with both predefined global constraints, and to capture and define new complex constraints by the modeller.

4.1 Global Constraints

There are many global constraints defined in MiniZinc for use in modelling. The definitive list is to be found in the documentation for the release, as the list is slowly growing. Below we discuss some of the most important global constraints

4.1.1 Alldifferent

The `alldifferent` constraint takes an array of variables and constrains them to take different values. A use of the `alldifferent` has the form

```
alldifferent(array[int] of var int:  x)
```

that is the argument is array of integer variables.

Alldifferent is one of the most studied and used global constraints in constraint programming. It is used to define assignment subproblems, and efficient global propagators for `alldifferent` exist `send-more-money.mzn` (Figure 11) and `sudoku.mzn` (Figure 12) are examples of models using `alldifferent`.

4.1.2 Cumulative

The `cumulative` constraint is used for describing cumulative resource usage.

```
cumulative(array[int] of var int: s, array[int] of var int: d,
           array[int] of var int: r, var int: b)
```

Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

The model in Figure 22 finds a schedule for moving furniture so that each piece of furniture has enough handlers (people) and enough trolleys available during the move. The available time, handlers and trolleys are given, and the data gives for each object the move duration, the number of handlers and the number of trolleys required. Using the data shown in Figure 23, the command

```
$ mzn moving.mzn moving.dzn
```

may result in the output

```
start = [0, 60, 60, 90, 120, 0, 15, 105]
end = 140
----------
```

Figures 24(a) and (b) show the requirements for handlers and trolleys at each time in the move for this solution.

4.1.3 Table

The `table` constraint enforces that tuple of variables takes a value from a set of tuples. Since there are no tuples in MiniZinc this is encoded using arrays. The usage of `table` has one of the forms

```
table(array[int] of var bool:  x, array[int, int] of bool:  t)
table(array[int] of var int:  x, array[int, int] of int:  t)
```

moving.mzn

```
include "cumulative.mzn";

int: n; % number of objects;
set of int: OBJECTS = 1..n;
array[OBJECTS] of int: duration; % duration to move
array[OBJECTS] of int: handlers; % number of handlers required
array[OBJECTS] of int: trolleys; % number of trolleys required

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;
var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers);
constraint cumulative(start, duration, trolleys, available_trolleys);

constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = ", show(start), "\nend = ", show(end), "\n"];
```

Figure 22: Model for moving furniture using `cumulative` (`moving.mzn`).

moving.dzn

```
n = 8;
% piano, fridge, double bed, single bed, wardrobe, chair, chair, table

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```

Figure 23: Data for moving furniture using `cumulative` (`moving.dzn`).

depending on whether the tuples are Boolean or integer. Enforces $x \in t$ where we consider x and each row in t to be a tuple, and t and to be a set of tuples.

The model in Figure 25 searches for balanced meals. Each meal item has a name (encoded as an integer), a kilojoule count, protein in grams, salt in milligrams, and fat in grams, as well as cost in cents. The relationship between these items is encoded using a `table` constraint. The model searches for a minimal cost meal which has a minimum kilojoule count min_energy, a minimum amount of protein $min\_protein$, maximum amount of salt max_salt and fat max_fat.

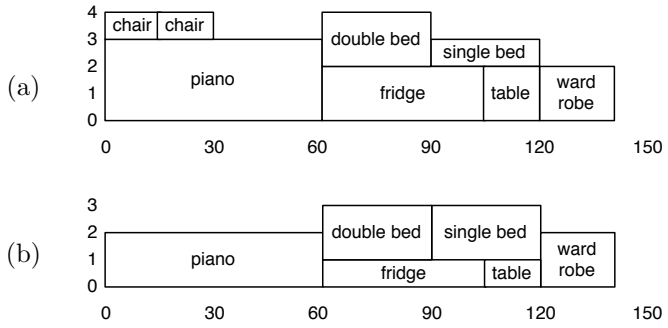

Figure 24: Histograms of usage of (a) handlers and (b) trolleys in the move.

4.1.4 Regular

The `regular` constraint is used to enforce that a sequence of variables takes a value defined by a finite automaton. The usage of `regular` has the form

```
regular(array[int] of var int: x, int: Q, int: S,
        array[int,int] of int: d, int: q0, set of int: F)
```

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $< 1..Q, 1..S >$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

Consider a nurse rostering problem. Each nurse is scheduled for each day as either: (d) on day shift, (n) on night shift, or (o) off. In each four day period a nurse must have at least one day off, and no nurse can be scheduled for 3 night shifts in a row. This can be encoded using the incomplete DFA shown in Figure 27. We can encode this DFA as having start state 1, final states 1 .. 6, and transition function

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

Note that state 0 in the table indicates an error state. The model shown in Figure 28 finds a schedule for *num_nurses* nurses over *num_days* days, where we require *req_day* nurses on day shift each day, and *req_night* nurses on night shift, and that each nurse takes at least *min_night* night shifts.

Running the command

```
$ mzn nurse.mzn nurse.dzn
```

finds a 10 day schedule for 7 nurses, requiring 3 on each day shift and 2 on each night shift, with a minimum 2 night shifts per nurse. A possible output is

```
d d d - d d d - n n
d d d - d d d - n n
d d - d d n - n d d
```

meal.mzn

```
% Planning a balanced meal
include "table.mzn";

int: min_energy;
int: min_protein;
int: max_salt;
int: max_fat;
set of FOODS: desserts;
set of FOODS: mains;
set of FOODS: sides;

set of int: FEATURES = 1..6;
int: name = 1;
int: energy = 2;
int: protein = 3;
int: salt = 4;
int: fat = 5;
int: cost = 6;

int: nf; % number of foods
set of int: FOODS = 1..nf;
array[FOODS] of string: n; % food names

array[FOODS,FEATURES] of int: dd; % food database

array[FEATURES] of var int: main;
array[FEATURES] of var int: side;
array[FEATURES] of var int: dessert;
var int: budget;

constraint main[name] in mains;
constraint side[name] in sides;
constraint dessert[name] in desserts;

constraint table(main, dd);
constraint table(side, dd);
constraint table(dessert, dd);

constraint main[energy] + side[energy] + dessert[energy] >= min_energy;
constraint main[protein] + side[protein] + dessert[protein] >= min_protein;
constraint main[salt] + side[salt] + dessert[salt] <= max_salt;
constraint main[fat] + side[fat] + dessert[fat] <= max_fat;
constraint budget = main[cost] + side[cost] + dessert[cost];

solve minimize budget;

output ["main = ",n[fix(main[name])],", side = ",n[fix(side[name])],
      ", dessert = ",n[fix(dessert[name])],", cost = ",show(budget), "\n"];
```

Figure 25: Model for meal planning using `table` constraint (`meal.mzn`).

```
n n - d n d - d d d
n - d n n - d d d -
- n n d - n n d - d
- - n n - - n n - -
```

meal.dzn

```
nf = 9;
n = ["icecream","banana","chocolate cake","lasagna",
     "steak","rice","chips","brocolli","beans"];

dd = [| 1, 1200,  50,  10, 120,  400     % icecream
      | 2,  800, 120,   5,  20,  120     % banana
      | 3, 2500, 400,  20, 100,  600     % chocolate cake
      | 4, 3000, 200, 100, 250,  450     % lasagna
      | 5, 1800, 800,  50, 100, 1200     % steak
      | 6, 1200,  50,   5,  20,  100     % rice
      | 7, 2000,  50, 200, 200,  250     % chips
      | 8,  700, 100,  10,  10,  125     % brocolli
      | 9, 1900, 250,  60,  90,  150 |]; % beans

min_energy = 3300;
min_protein = 500;
max_salt = 180;
max_fat = 320;
desserts = {1,2,3};
mains = {4,5,9};
sides = {6,7,8};
```

Figure 26: Data for meal planning defining the `table` used (`meal.dzn`).

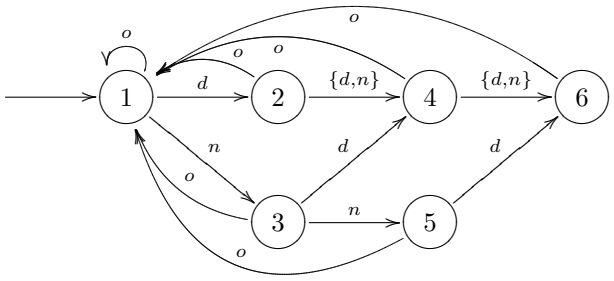

Figure 27: A DFA determining correct rosters.

4.2 Defining Predicates

One of the most powerful modelling features of MiniZinc is the ability for the modeller to define their own high-level constraints. This allows them to abstract and modularise their model. It also allows re-use of constraints in different models and allows the development of application specific libraries defining the standard constraints and types.

We start with a simple example, revisiting the job shop scheduling problem from the previous section. The model is shown in Figure 29. The item of interest is the `predicate` item:

```
predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;
```

This defines a new constraint that enforces that a task with start time `s1` and duration `d1` does not overlap with a task with start time `s2` and duration `d2`. This

nurse.mzn

```
% Simple nurse rostering
include "regular.mzn";

int: num_nurses;
set of int: NURSES = 1..num_nurses;
int: num_days;
set of int: DAYS = 1..num_days;

int: req_day;
int: req_night;
int: min_night;

int: S = 3;
set of int: SHIFTS = 1..S;
int: d = 1;
int: n = 2;
int: o = 3;
array[SHIFTS] of string: name = ["d","n","-"];

int: Q = 6;
int: q0 = 1;
set of int: STATES = 1..Q;
array[STATES,SHIFTS] of int: t =
     [| 2, 3, 1    % state 1
      | 4, 4, 1    % state 2
      | 4, 5, 1    % state 3
      | 6, 6, 1    % state 4
      | 6, 0, 1    % state 5
      | 0, 0, 1|]; % state 6

array[NURSES,DAYS] of var SHIFTS: roster;

constraint forall(j in DAYS)(
               sum(i in NURSES)(bool2int(roster[i,j] == d)) == req_day /\
               sum(i in NURSES)(bool2int(roster[i,j] == n)) == req_night
           );

constraint forall(i in NURSES)(
               regular( [roster[i,j] | j in DAYS], Q, S, t, q0, STATES) /\
               sum(j in DAYS)(bool2int(roster[i,j] == n)) >= min_night
           );

solve satisfy;

output [ name[fix(roster[i,j])] ++ if j==num_days then "\n" else " " endif
       | i in NURSES, j in DAYS ];
```

Figure 28: Model for nurse rostering using `regular` constraint (`nurse.mzn`).

can now be used inside the model anywhere any other Boolean expression (involving decision variables) can be used.

As well as predicates the modeller can define new constraints that only involve parameters. Unlike predicates these can be used inside the test of a conditional expression. These are defined using the key word `test`. For example

jobshop2.mzn

```
int: jobs;                                      % no of jobs
int: tasks;                                     % no of tasks per job
array [1..jobs,1..tasks] of int: d;             % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
             (d[i,j]);                          % total duration
array [1..jobs,1..tasks] of var 0..total: s;    % start times
var 0..total: end;                              % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        forall(i,k in 1..jobs where i < k) (
            no_overlap(s[i,j], d[i,j], s[k,j], d[k,j])
        )
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show(s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```

Figure 29: Model for job shop scheduling using predicates (`jobshop2.mzn`).

```
test even(int:x) = x mod 2 = 0;
```

4.3 Reflection Functions

To help write generic tests and predicates, various reflection functions return information about array index sets, var set domains and decision variable ranges. Those for index sets are `index_set(`⟨*1-D array*⟩`)`, `index_set_1of2(`⟨*2-D array*⟩`)`, `index_set_2of2(`⟨*2-D array*⟩`)` and so on for higher dimensional arrays.

A better model of the job shop conjoins all the non-overlap constraints for a single machine into a single disjunctive constraint. An advantage of this approach is that while we may initially model this simply as a conjunction of non-overlap, if the underlying solver has a better approach to solving disjunctive constraints we can use that instead, with minimal changes to our model. The model is shown in Figure 30.

The `disjunctive` constraint takes an array of start times for each task and an array of their durations and makes sure that only one task is active at any one time. We define the disjunctive constraint as a predicate with signature

```
predicate disjunctive(array[int] of var int:s, array[int] of int:d);
```

jobshop3.mzn

```
include "disjunctive.mzn";

int: jobs;                                      % no of jobs
int: tasks;                                     % no of tasks per job
array [1..jobs,1..tasks] of int: d;             % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
             (d[i,j]);                          % total duration
array [1..jobs,1..tasks] of var 0..total: s;    % start times
var 0..total: end;                              % total end time


constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        disjunctive([s[i,j] | i in 1..jobs], [d[i,j] | i in 1..jobs])
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show(s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```

Figure 30: Model for job shop scheduling using `disjunctive` predicate (`jobshop3.mzn`).

We can use the disjunctive constraint to define the non-overlap of tasks as shown in Figure 30. We assume a definition for the `disjunctive` predicate is given by the file `disjunctive.mzn` which is included in the model. If the underlying system supports `disjunctive` directly, it will include a file `disjunctive.mzn` in its globals directory (with contents just the signature definition above). If the system we are using does not support disjunctive directly we can give our own definition by creating the file `disjunctive.mzn`. The simplest implementation simply makes use of the `no_overlap` predicate defined above. A better implementation is to make use of a global `cumulative` constraint assuming it is supported by the underlying solver. Figure 31 shows an implementation of `disjunctive`. Note how we use the `index_set` reflection function to (a) check that the arguments to `disjunctive` make sense, and (b) construct the array of resource utilisations of the appropriate size for `cumulative`. Note also that we use a ternary version of `assert` here

Note that assert expressions are lazy in the third argument, that is if the first argument is false they will not be evaluated. Hence they can be used for checking

```
predicate lookup(array[int] of var int:x, int: i, var int: y) =
    assert(i in index_set(x), "index out of range in lookup"
           y = x[i]
    );
```

This code will not evaluate $x[i]$ if i is out of the range of the array x.

disjunctive.mzn

```
include "cumulative.mzn";

predicate disjunctive(array[int] of var int:s, array[int] of int:d) =
          assert(index_set(s) == index_set(d), "disjunctive: " ++
              "first and second arguments must have the same index set",
            cumulative(s, d, [ 1 | i in index_set(s) ], 1)
          );
```

Figure 31: Defining a `disjunctive` predicate using `cumulative` (`disjuntive.mzn`).

Predicate definitions:
Predicates are defined by a statement of the form

`predicate` ⟨pred-name⟩ `(` ⟨arg-def⟩, . . . , ⟨arg-def⟩ `)` = ⟨bool-exp⟩

The ⟨pred-name⟩ must be a valid MiniZinc identifier, and each ⟨arg-def⟩ is a valid MiniZinc type declaration. One relaxation of argument definitions is that the index types for arrays can be unbounded written `int`.
We also introduce a new form of the `assert` command for use in predicates.

`assert (` ⟨bool-exp⟩, ⟨string-exp⟩, ⟨exp⟩ `)`

The type of the `assert` expression is the same as the type of the last argument. The `assert` expression checks whether the first argument is false, and if so prints the second argument string. If the first argument is true it returns the third argument.

4.4 Local Variables and Domain Reflection Functions

It is often useful to introduce *local variables* in a test or predicate. The `let` expression allows you to do so. It can be used to introduce both decision variables and parameters, but parameters must be initialised. For example:

```
var s..e: x;
let {int: l = s div 2, int: u = e div 2, var l .. u: y} in x = 2*y
```

introduces parameters l and u and variable y. While most useful in predicate and test definitions, let expressions can also be used in other expressions, for example for eliminating common subexpressions:

```
constraint let { var int: s = x1 + x2 + x3 + x4 } in
           l <= s /\ s <= u;
```

Local variables can be used anywhere and can be quite useful, for simplifying complex expressions. Figure 32 gives a revised version of the wedding model, using local variables to define the objective function, rather than adding lots of variables to the model explicitly.

One limitation is that predicates containing decision variables that are not initialised in the declaration cannot be used inside a negative context. The following is illegal

```
predicate even(var int:x) =
          let { var int: y } in x = 2 * y;
```

wedding2.mzn

```
set of int: Guests = 1..12;
int: bride = 1; int: groom = 2; int: bestman = 3;
int: bridesmaid = 4; int: bob = 5; int: carol  = 6;
int: ted = 7; int: alice = 8; int: ron = 9;
int: rona = 10; int: ed = 11; int: clara = 12;
array[Guests] of string: name = ["bride","groom","bestman",
     "bridesmaid","bob","carol","ted","alice","ron","rona","ed","clara"];
set of int: Seats = 1..12;

set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];

set of int: Males = {groom, bestman, bob, ted, ron,ed};
set of int: Females = {bride, bridesmaid, carol, alice, rona, clara};

array[Guests] of var Seats: pos;

include "alldifferent.mzn";

constraint alldifferent(pos);

constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );

constraint pos[ed] != 1 /\ pos[ed] != 6 /\ pos[ed] != 7 /\ pos[ed] != 12;
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);

solve maximize sum(h in Hatreds)(
          let {  var Seats: p1 = pos[h1[h]],
                 var Seats: p2 = pos[h2[h]],
                 var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6) } in
          same * abs(p1 - p2) +
          (1-same) * (abs(13 - p1 - p2) + 1));

output [ name[g] ++ " " | s in Seats, g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

Figure 32: Using local variables to define a complex objective function (`wedding2.mzn`)

```
constraint not even(z);
```

Example: Other important reflection functions are those that allow us to access the domains of variables. The expression `lb(x)` returns a value that is lower than or equal to any value that x may take in a solution of the problem. Usually it will just be the declared lower bound of x. If x is declared as a non-finite type, e.g. simply `var int` then it is an error. Similarly the expression `dom(x)` returns a (non-strict) superset of the possible values of x in any solution of the problem. Again it is usually the declared values, and again if it is not declared as finite then there is an error.

For example, the model show in Figure 33 may output

Let expressions:
Local variables can be introduced in any expression with a *let expression* of the form:

`let` { ⟨var-dec⟩, ... ⟨var-dec⟩ } `in` ⟨*exp*⟩

The variable declarations ⟨var-dec⟩ can contain both decision variables and parameters. Parameters must be initialised.
Note that local variables cannot occur in predicates that appear in a negative context

reflection.mzn

```
var -10..10: x;
constraint x in 0..4;
int: y = lb(x);
set of int: D = dom(x);
solve satisfy;
output ["y = ", show(y), "\nD = ", show(D), "\n"];
```

Figure 33: Using reflection predicates (`reflection.mzn`)

```
y = -10
D = {-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10}
----------
```

or

```
y = 0
D = {0, 1, 2, 3, 4}
----------
```

or any answer with $-10 \leq y \leq 0$ and $\{0, \ldots, 4\} \subseteq D \subseteq \{-10, \ldots, 10\}$.

Variable domain reflection expressions should be used in a manner where they are correct for any safe approximations, but note this is not checked! For example the additional code

```
var -10..10: z;
constraint z <= y;
```

is not a safe usage of the domain information. Since using the tighter (correct) approximation leads to more solutions then the weaker initial approximation.

The combinations of predicates, local variables and domain reflection allows the definition of complex global constraints by decomposition. We can define the time based decomposition of the `cumulative` constraint using the code shown in Figure 34.

The decomposition uses `lb` and `ub` to determine the set of times `times` over which tasks could range. It then asserts for each time t in `times` that the sum of resources for the active tasks at time t is less than the bound b.

4.5 Scope

It is worth briefly mentioning the scope of declarations in MiniZinc . MiniZinc has a single namespace, so all variables appearing in declarations are visible in every

Domain reflection:
There are reflection functions to interrogate the possible values of expressions containing variables:

- `dom (` ⟨exp⟩ `)`: returns a safe approximation to the possible values of the expression.
- `lb (` ⟨exp⟩ `)`: returns a safe approximation to the lower bound value of the expression.
- `ub (` ⟨exp⟩ `)`: returns a safe approximation to the upper bound value of the expression.

The expressions for `lb` and `ub` can only be of types `int`, `bool`, `float` or `set of int`. For `dom` the type cannot be `float`. If one of the variables appearing in *exp* has a non-finite declared type (e.g. `var int` or `var float` type) then an error can occur.
There are also versions that work directly on arrays of expressions (with similar restrictions):

- `dom_array (` ⟨array-exp⟩ `)`: returns a safe approximation to the union of all possible values of the expressions appearing in the array.
- `lb_array (` ⟨array-exp⟩ `)`: returns a safe approximation to the lower bound of all expressions appearing in the array.
- `ub_array (` ⟨array-exp⟩ `)`: returns a safe approximation to the upper bound of all expressions appearing in the array.

expression in the model. MiniZinc introduces locally scoped variables in a number of ways:

- as iterator variables in comprehension expressions
- using `let` expressions
- as predicate arguments

Any local scoped variable overshadows the outer scoped variables of the same name.

For example, in the model shown in Figure 35 the `x` in `-x <= y` is the global `x`, the `x` in `even(x)` is the iterator `x in 1..u`, while the `y` in the disjunction is the second argument of the predicate.

cumulative.mzn

```
%-----------------------------------------------------------------------%
% Requires that a set of tasks given by start times 's', durations 'd',
% and resource requirements 'r', never require more than a global
% resource bound 'b' at any one time.
% Assumptions:
% - forall i, d[i] >= 0 and r[i] >= 0
%-----------------------------------------------------------------------%
predicate cumulative(array[int] of var int: s,
                     array[int] of var int: d,
                     array[int] of var int: r, var int: b) =
    assert(index_set(s) == index_set(d) /\
           index_set(s) == index_set(r),
        "cumulative: the array arguments must have identical index sets",
    assert(lb_array(d) >= 0 /\ lb_array(r) >= 0,
        "cumulative: durations and resource usages must be non-negative",
            let {
                set of int: times =
                  min([ lb(s[i]) | i in index_set(s) ]) ..
                  max([ ub(s[i]) + ub(d[i]) | i in index_set(s) ])
                }
            in
               forall( t in times ) (
                  b >= sum( i in index_set(s) ) (
                     bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]
                  )
               )
        )
    );
```

Figure 34: Defining a `cumulative` predicate by decomposition (`cumulative.mzn`).

scope.mzn

```
int: x = 3;
int: y = 4;
predicate smallx(var int:y) = -x <= y /\ y <= x;
predicate p(int: u, var bool: y) =
      exists(x in 1..u)(y \/ smallx(x));
constraint p(x,false);
solve satisfy;
```

Figure 35: A model for illustrating scopes of variables (`scope.mzn`)

5 Search

By default in MiniZinc there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver. But sometimes, particularly for combinatorial integer problems, we may want to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy. Note that the search strategy is *not* really part of the model. Indeed it is not required that each solver implements all possible search strategies. MiniZinc uses a consistent approach to communicating extra information to the constraint solver using *annotations*.

5.1 Finite Domain Search

Search in a finite domain solver involves examining the remaining possible values of variables and choosing to constrain some variables further. The search then adds a new constraint that restricts the remaining values of the variable (in effect guessing where the solution might lie), and then applies propagation to determine what other values are still possible in solutions. In order to guarantee completeness, the search leaves another choice which is the negation of the new constraint. The search ends either when the finite domain solver detects that all constraints are satisfied, and hence a solution has been found, or that the constraints are unsatisfiable. When unsatisfiability is detected the search must proceed down a different set of choices. Typically finite domain solvers use *depth first search* where they undo the last choice made and then try to make a new choice.

nqueens.mzn

```
int: n;
array [1..n] of var 1..n: q; % queen is column i is in row q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                          % distinct rows
constraint alldifferent([ q[i] + i | i in 1..n]); % distinct diagonals
constraint alldifferent([ q[i] - i | i in 1..n]); % upwards + downwards

solve :: int_search(q, first_fail, indomain_min, complete)
      satisfy;

output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

Figure 36: Model for n-queens (`nqueens.mzn`).

A simple example of a finite domain problem is the n queens problem which requires that we place n queens on an $n \times n$ chessboard so that none can attack another. The variable $q[i]$ records in which row the queen in column i is placed. The `alldifferent` constraints ensure that no two queens are on the same row, or diagonal. A typical (partial) search tree for $n = 9$ is illustrated in the left of Figure 37. We first set $q[1] = 1$, this removes values from the domains of other variables, so that e.g. $q[2]$ cannot take the values 1 or 2. We then set $q[2] = 3$, this further removes values from the domains of other variables. We set $q[3] = 5$ (its earliest possible value). The state of the chess board after these three decisions is shown in Figure 37(a) where the queens indicate the position of the queens fixed already and the stars indicate positions where we cannot place a queen since it

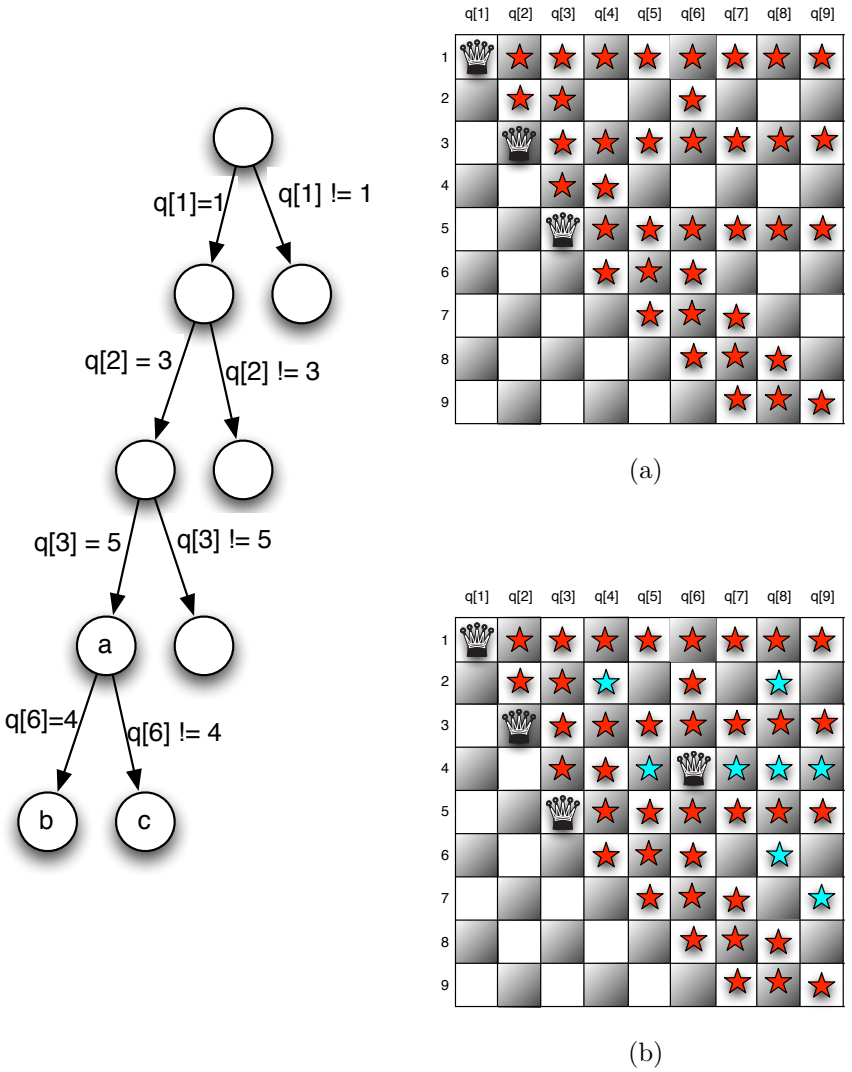

Figure 37: Partial search trees for 9 queens: (a) the state after the addition of $q[1] = 1$, $q[2] = 4$, $q[3] = 5$ (b) the initial propagation on adding further $q[6] = 4$.

would be able to take an already placed queen.

A search strategy determines which choices to make. The decisions we have made so far follow the simple strategy of picking the first variable which is not fixed yet, and try to set it to its least possible value. Following this strategy the next decision would be $q[4] = 7$. An alternate strategy for variable selection is to choose the variable whose current set of possible values (*domain*) is smallest. Under this so called *first-fail* variable selection strategy the next decision would be $q[6] = 4$. If we make this decision, then initially propagation removes the additional values shown in Figure 37(b). But this leaves only one value for $q[8]$, $q[8] = 7$, so this is forced, but then this leaves only one possible value for $q[7]$ and $q[9]$, that is 2. Hence a constraint must be violated. We have detected unsatisfiability, and the solver must backtrack undoing the last decision $q[6] = 4$ and adding its negation $q[6] \neq 4$ (leading us to state (c) in the tree in Figure 37) which forces $q[6] = 8$.

This removes some values from the domain and then we again reinvoke the search strategy to decide what to do.

Many finite domain searches are defined in this way: choose a variable to constrain further, and then choose how to constrain it further.

5.2 Search Annotations

Search annotations in MiniZinc specify how to search in order to find a solution to the problem. The annotation is attached to the solve item, after the keyword `solve` The search annotation

```
solve :: int_search(q, first_fail, indomain_min, complete)
      satisfy;
```

appears on the solve item. Annotations are attached to parts of the model using the connector `::`. This search annotation means that we should search by selecting from the array of integer variables `q`, the variable with the smallest current domain (this is the `first_fail` rule), and try setting it to its smallest possible value (`indomain_min` value selection), looking across the entire search tree (`complete` search).

Basic search annotations:

There are four basic search annotations corresponding to different basic variable types:

- `int_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var int`, *varchoice* is a variable choice annotation discusses below, *constrainchoice* is a choice of how to constrain a variable, discussed below, and *strategy* is a search strategy which we will assume for now is `complete`.
- `bool_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var bool` and the rest are as above.
- `set_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var set of int` and the rest are as above.

Example variable choice annotations are: `input_order` choose in order from the array, `first_fail` choose the variable with the smallest domain size, and `smallest` choose the variable with the smallest value in its domain.
Example ways to constraint a variable are: `indomain_min` assign the variable its smallest domain value, `indomain_median` assign the variable its median domain value, `indomain_random` assign the variable a random value from its domain, and `indomain_split` bisect the variables domain excluding the upper half.
Strategy is almost always `complete` for complete search. For a complete list of variable and constraint choice annotations see the FlatZinc specification in the MiniZinc reference documentation.

We can construct more complex search strategies using search constructor annotations. There is only one such annotation at present.

```
seq_search([ search_ann, ..., search_ann ])
```

The sequential search constructor first undertakes the search given by the first annotation in its list, when all variables in this annotation are fixed it undertakes the second search annotation, etc until all search annotations are complete.

Consider the jobshop scheduling model shown in Figure 30. We could replace the solve item with

```
solve :: seq_search([
             int_search(s, smallest, indomain_min, complete),
             int_search([end], input_order, indomain_min, complete)])
      minimize end
```

which tries to set start times `s` by choosing the job that can start earliest and setting it to that time. When all start times are complete the end time `end` may not be fixed. Hence we set it to its minimal possible value.

5.3 Annotations

Annotations are a first class object in MiniZinc . We can declare new annotations in a model, and declare and assign to annotation variables.

Annotations:
Annotations have a type `ann`. You can declare an annotation parameter (with optional assignment)

`ann` : ⟨ident⟩ [= ⟨ann-expr⟩] ;

and assign to an annotation variable just as any other parameter.
Expressions, variable declarations, and `solve` items can all be annotated using the `::` operator.
We can declare a new annotation using the annotation item

`annotation` ⟨annotation-name⟩(⟨arg-def⟩, . . . , ⟨arg-def⟩) ;

The program in Figure 38 illustrates the use of annotation declarations, annotations and annotation variables. We declare a new annotation `bitdomain` which is meant to tell the solver that variables domains should be represented via bit arrays of size `nwords`. The annotation is attached to the declarations of the variables `q`. Each of the `alldifferent` constraints is annotated with the built in annotation `domain` which instructs the solver to use the domain propagating version of `alldifferent` if it has one. An annotation variable `search_ann` is declared and used to define the search strategy. We can give the value to the search strategy in a separate data file.

Example search annotations might be the following (where we imagine each line is in a separate data file)

```
search_ann = int_search(q, input_order, indomain_min, complete);
search_ann = int_search(q, input_order, indomain_median, complete);
search_ann = int_search(q, first_fail, indomain_min, complete);
search_ann = int_search(q, first_fail, indomain_median, complete);
```

The first just tries the queens in order setting them to the minimum value, the second tries the queens variables in order, but sets them to their median value, the third tries the queen variable with smallest domain and sets it to the minimum value, and the final strategy tries the queens variable with smallest domain setting it to its median value.

nqueens-ann.mzn

```
annotation bitdomain(int:nwords);

include "alldifferent.mzn";

int: n;
array [1..n] of var 1..n: q :: bitdomain(n div 32);

constraint alldifferent(q) :: domain;
constraint alldifferent([ q[i] + i | i in 1..n]) :: domain;
constraint alldifferent([ q[i] - i | i in 1..n]) :: domain;

ann: search_ann;

solve :: search_ann satisfy;

output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

Figure 38: Annotated model for n-queens (`nqueens-ann.mzn`).

Different search strategies can make a significant difference in how easy it is to find solutions. A small comparison of the number of choices made to find the first solution of the n-queens problems using the 4 different search strategies is shown in the table below (where — means more than 100,000 choices). Clearly the right search strategy can make a significant difference.

n	input-min	input-median	ff-min	ff-median
10	28	15	16	20
15	248	34	23	15
20	37330	97	114	43
25	7271	846	2637	80
30	—	385	1095	639
35	—	4831	—	240
40	—	—	—	236

6 Effective Modelling Practices in MiniZinc

MiniZinc is a complex enough modelling language that there can be multiple ways to model the same problem, some of which generate models which are efficient to solve, and some of which are not. In general it is very hard to tell apriori which models are the most efficient for solving a particular problem, and indeed it may critically depend on the underlying solver used, and search strategy. In this chapter we concentrate on modelling practices that avoid inefficiency in generating models and generated models.

6.1 Variable Bounds

Finite domain propagation engines, which are the principle type of solver targeted by MiniZinc are more effective the tighter the bounds on the variables involved. They can also behave badly with problems which have subexpressions that take large integer values, since they may implicitly limit the size of integer variables.

unbounded.mzn

```
var int: a;
var int: b;

constraint 200*a + 150 * b <= 4000;
constraint 100*a + 250 * b >= 0;

solve satisfy;

output ["a = ",show(a), " b = ", show(b), "\n"];
```

Figure 39: A model with unbounded integers (`unbounded.mzn`).

The result of executing

```
$ mzn unbounded.mzn
```

for the default `fd` solver is the error message

```
evaluation of model would result in integer overflow.
```

This is because the `var int` variables get default bounds $-1,000,000..1,000,000$ leading to intermediate expressions which are to big to be represented in the default bounds. Changing the variable declarations to `var -1000..1000` will lead to solutions being found immediately. Note that using the mixed integer solver

```
$ mzn -b mip unbounded.mzn
```

immediately returns an answer `a = 0 b = 0`.

Note that even models where all variables are bounded, may introduce intermediate expressions that are too large for the solver.

The grocery problem shown in Figure 40 builds a large product, which requires bigger bounds than the default, even though all the (declared) variables are tightly bounded. This may result in the solver detecting failure since there is not solution in the default bounds. Executing

```
$ mzn grocery.mzn
```

reports

grocery.mzn

```
var 0..711: item1;
var 0..711: item2;
var 0..711: item3;
var 0..711: item4;

constraint item1 + item2 + item3 + item4 == 711;
constraint item1 * item2 * item3 * item4 == 711 * 100 * 100 * 100;

% symmetry breaking
constraint item1 <= item2 /\ item2 <= item3 /\ item3 <= item4;

solve satisfy;

output ["{", show(item1), ",", show(item2), ",", show(item3), ",",
         show(item4),"}\n"];
```

Figure 40: A model with large intermediate values (`grocery.mzn`).

```
mzn2fzn:
  error:
  grocery.mzn:18
  In constraint.
  Model inconsistency detected.
mzn:  flattening failed
```

which shows that flattening the large product constraint resulted in an inconsistent constraint (because of the default bounds).

Bounding variables:
Always try to use bounded variables in models. When using `let` declarations to introduce new variables, always try to define them with correct and tight bounds. This will make your model more efficient, and avoid the possibility of unexpected overflows. One exception is when you introduce a new variable which is immediately defined as equal to an expression. Usually MiniZinc will be able to infer effective bounds from the expression.

6.2 Unconstrained Variables

Sometimes when modelling it is easier to introduce more variables than actually required to model the problem.

Consider the model for Golomb rulers shown in Figure 41. A Golomb ruler of n marks is one where the absolute differences between any two marks are different. It creates a two dimensional array of difference variables, but only uses those of the form `diff[i,j]` where $i > j$. Running the model with data file `golomb1.dzn` containing `n = 4; m = 6;` by

```
$ mzn golomb.mzn golomb.dzn
```

results in output

```
marks = [0, 2, 5, 6]
----------
```

golomb.mzn

```
include "alldifferent.mzn";

int: n; % number of marks on ruler
int: m; % max length of ruler

array[1..n] of var 0..m: mark;
array[1..n,1..n] of var 0..m: diffs;

constraint mark[1] = 0;
constraint forall ( i in 1..n-1 ) ( mark[i] < mark[i+1] );
constraint forall (i,j in 1..n where i > j)          % (diff)
                  (diffs[i,j] = mark[i] - mark[j]);  % (diff)
constraint alldifferent([ diffs[i,j] | i,j in 1..n where i > j]);
constraint diffs[2,1] < diffs[n,n-1];  % symmetry break

solve satisfy;

output ["marks = ",show(mark),"\n"];
```

Figure 41: A model for Golomb rulers with unconstrained variables (`golomb.mzn`).

and everything seems fine with the model. But if we ask for all solutions using `$ mzn -a golomb.mzn golomb.dzn` we are presented with a never ending list of the same solution!

What is going on? In order for the finite domain solver to finish it needs to fix all variables, including the variables `diff[i,j]` where $i \leq j$, which means there are countless ways of generating a solution, simply by changing these variables to take arbitrary values.

We can avoid problems with unconstrained variables, by modifying the model so that they are fixed to some value. For example replacing the lines marked `(diff)` in Figure 41 to

```
constraint forall (i,j in 1..n)
                  (diffs[i,j] = if (i > j) then mark[i] - mark[j]
                                else 0 endif);
```

ensures that the extra variables are all fixed to 0. With this change the solver returns just one solution.

The real danger of unconstrained search variables is when they are searched upon. Since they are irrelevant to the problem they can explode the search, for no purpose.

Consider the unsatisfiable model shown in Figure 42 where the Boolean variables `b` are unconstrained. Executing

```
$ mzn -S unconstrained.mzn
```

which prints statistics shows that search requires 3s and searches 490,000 choices. Removing the useless Boolean variables means the search is instant and searches 14 choices.

6.3 Effective Generators

Imagine we want to count the number of triangles (K_3 subgraphs) appearing in a graph. Suppose the graph is defined by an adjacency matrix: $adj[i, j]$ is true if nodes i and j are adjacent. We might write

unconstrained.mzn

```
include "alldifferent.mzn";

array[1..15] of var bool: b;
array[1..4] of var 1..10: x;

constraint alldifferent(x) /\ sum(i in 1..4)(x[i]) = 9;

solve satisfy;
```

Figure 42: A model with unconstrained variables (`unconstrained.mzn`).

Unconstrained Variables:
Models should never have unconstrained variables. Sometimes it is difficult to model without unnecessary variables. If this is the case add constraints to fix the unnecessary variables, so they cannot influence the solving.

```
int: count = sum [ 1 | i,j,k in NODES where i < j  /\ j < k
                       /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]];
```

which is certainly correct, but it examines all triples of nodes. If the graph is sparse we can do better by realising that some tests can be applied as soon as we select i and j.

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
          sum([1 | k in NODES where j < k /\ adj[i,k] /\ adj[j,k]]));
```

6.4 Redundant Constraints

The form of a model will effect how well the constraint solver can solve it. In many cases adding constraints which are redundant, i.e. are logically implied by the existing model, may improve the search for solutions by making more information available to the solver earlier.

Consider the magic series problem from Section 3.4. Running this on for $n = 16$ as follows:

```
$ mzn --all-solutions --statistics magic-series.mzn -D "n=16;"
```

might result in output

```
s = [12, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0];
----------
==========
```

and the statistics showing 174 choice points required.

We can add redundant constraints to the model. Since each number in the sequence counts the number of occurrences of a number we know that they sum up to n. Similarly we know that the sum of $s[i] \times i$ must also add up to n because the sequence is magic. Adding these constraints to our model gives the model in Figure 43.

Running the same problem as before

```
$ mzn --all-solutions --statistics magic-series2.mzn -D "n=16;"
```

results in the same output, but with statistics showing just 13 choicepoints explored. The redundant constraints have allowed the solver to prune the search much earlier.

magic-series2.mzn

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

constraint sum(i in 0..n-1)(s[i]) = n;
constraint sum(i in 0..n-1)(s[i] * i) = n;

solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

Figure 43: Model solving the magic series problem with redundant constraints (`magic-series2.mzn`).

6.5 Modelling Choices

There are many ways to model the same problem in MiniZinc , although some may be more natural than others. Different models may have very different efficiency of solving, and worse yet, different models may be better or worse for different solving backends. There are however some guidelines for usually producing better models:

Choosing between models:
The better model is likely to have some of the following features

- smaller number of variables, or at least those that are not functionally defined by other variables
- smaller domain sizes of variables
- more succinct, or direct, definition of the constraints of the model
- uses global constraints as much as possible

In reality all this has to be tempered by how effective the search is for the model. Usually the effectiveness of search is hard to judge except by experimentation.

Consider the problem of finding permutations of n numbers from 1 to n such that the differences between adjacent numbers also form a permutation of numbers 1 to $n-1$. Note that the u variables are functionally defined by the x variables so the raw search space is n^n The obvious way to model this problem is shown in Figure 44

In this model the array x represents the permutation of the n numbers and the constraints are naturally represented using `alldifferent`. Running the model

```
$ mzn -all-solutions --statistics allinterval.mzn -D "n=10;"
```

finds all solutions in 84598 choice points and 3s.

An alternate model is model uses array y where $y[i]$ gives the position of the number i in the sequence. We also model the positions of the differences using variables v. $v[i]$ is the position in the sequence where the absolute difference i occurs. If the value of $y[i]$ and $y[j]$ differ by one where $j > i$, meaning the positions

allinterval.mzn

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: x;      % sequence of numbers
array[1..n-1] of var 1..n-1: u;  % sequence of differences

constraint alldifferent(x);
constraint alldifferent(u);
constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

solve :: int_search(x, first_fail, indomain_min, complete)
      satisfy;
output ["x = ",show(x),"\n"];
```

Figure 44: A natural model for the all interval series problem "prob007" in CSPlib. (`allinterval.mzn`).

are adjacent, then $v[j-i]$ is constrained to be the earliest of these positions. We can add two redundant constraints to this model: since we know that a difference of $n-1$ must result, we know that the positions of 1 and n must be adjacent $|y[1]-y[n]|=1$, which also tell us that the position of difference $n-1$ is the earlier of $y[1]$ and $y[n]$, i.e. $v[n-1] = min(y[1], y[n])$. With this we can model the problem as shown in Figure 45. The output statement recreates the original sequence x from the array of positions y.

The inverse model has the same size as the original model, in terms of number of variables and domain sizes. But the inverse model has a much more indirect way of modelling the relationship between y and v variables as opposed to the relationship between x and u variables. Hence we might expect the original model to be better.

The command

```
$ mzn --all-solutions --statistics allinterval2.mzn -D "n=10;"
```

finds all the solutions in 75536 choice points and 18s. Interesting, although the model is not as succinct here, the search on the y variables is better than searching on the x variables. The lack of succinctness means that even though the search requires less choice it is substantially slower.

6.6 Multiple Modelling and Channels

When we have two models for the same problem it may be useful to use both models together by tying the variables in the two models together, since each can give different information to the solver.

Figure 46 gives a dual model combining features of `allinterval.mzn` and `allinterval2.mzn`. The beginning of the model is taken from `allinterval.mzn`. We then introduce the y and v variables from `allinterval2.mzn`. We tie the variables together using the global `inverse`: $inverse(x, y)$ holds if y is the inverse function of x (and vice versa) that is $x[i] = j \Leftrightarrow y[j] = i$. A definition is shown in Figure 47. The model does not include the constraints relating the y and v variables, they are redundant (and indeed propagation redundant so they do not add information for a propagation solver. The `alldifferent` constraints are also missing since they are made redundant (and propagation redundant) by the inverse

allinterval2.mzn

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: y;  % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint alldifferent(y);
constraint alldifferent(v);
constraint forall(i,j in 1..n where i < j)(
                          (y[i] - y[j] = 1 -> v[j-i] = y[j]) /\
                      (y[j] - y[i] = 1 -> v[j-i] = y[i])
           );

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output [ "x = [",] ++
       [ show(i) ++ if j == n then "]\n;" else ", " endif
         | j in 1..n, i in 1..n where j == fix(y[i]) ];
```

Figure 45: An inverse model for the all interval series problem "prob007" in CSPlib. (`allinterval2.mzn`).

allinterval3.mzn

```
include "inverse.mzn";

int: n;

array[1..n] of var 1..n: x;  % sequence of numbers
array[1..n-1] of var 1..n-1: u;  % sequence of differences

constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

array[1..n] of var 1..n: y;  % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint inverse(x,y);
constraint inverse(u,v);

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output ["x = ",show(x),"\n"];
```

Figure 46: A dual model for the all interval series problem "prob007" in CSPlib. (`allinterval3.mzn`).

inverse.mzn

```
predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(invf))(invf[j] in index_set(f)) /\
    forall(i in index_set(f))(
        f[i] in index_set(invf) /\
        forall(j in index_set(invf))(j == f[i] <-> i == invf[j])
    );
```

Figure 47: A definition of the `inverse` global constraint. (`inverse.mzn`).

constraints. The only constraints are the relationships of the x and u variables and the redundant constraints on y and v.

One of the benefits of the dual model is that there is more scope for defining different search strategies. Running the dual model,

```
$ mzn -all-solutions --statistics allinterval3.mzn -D "n=10;"
```

which note uses the search strategy of the inverse model, labelling the y variables, finds all solutions in 1714 choice points and 0.5s. Note that running the same model with labelling on x variables requires 13142 choice points and 1.5s.

A MiniZinc Keywords

Note that since MiniZinc shares a parser with Zinc, all the Zinc keywords are also not usable as MiniZinc identifiers. The keywords are:

`ann`, `annotation`, `any`, `array`, `assert`, `bool`, `constraint`, `enum`, `float`, `function`, `in`, `include`, `int`, `list`, `of`, `op`, `output`, `minimize`, `maximize`, `par`, `predicate`, `record`, `set`, `solve`, `string`, `test`, `tuple`, `type` `var`, `where`.