Topic 2: Basic Modelling
(Version of 5th December 2018)

Pierre Flener and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

Outline

Outline

MiniZinc Model

A MiniZinc model may comprise the following items:

- Parameter declarations
- Variable declarations
- Predicate and function definitions
- Constraints
- Objective
- Output command

Types for Parameters

MiniZinc is strongly typed. The parameter types are:

- `int`: integer
- `bool`: Boolean
- `enum`: enumeration
- `float`: floating-point number
- `string`: string of characters
- `set of` τ: set of elements of type τ, which is `int`, `bool`, `enum`, `float`, or `string`
- `array[`ρ`] of` τ: possibly multidimensional array of elements of type τ, which is not an array; each range in ρ is an enumeration or an integer interval α`..`β

Example

The parameter declaration `int: n` declares an integer parameter of identifier `n`. One can also write `par int: n` in order to emphasise that `n` is a parameter.

Types for (Decision) Variables

Decision variables are implicitly existentially quantified: the aim is to find satisfying (optimal) values in their domains.

The variable types for decision variables are:

- `int`: integer
- `bool`: Boolean
- `enum`: enumeration
- `float`: floating-point number (not used in this course)
- `set of enum` and `set of int`: set

A possibly multidimensional `array` can be declared to have variables of any variable type, but it is itself not a variable.

Example

The variable declaration `var int: n` declares a decision variable of domain `int` and identifier `n`.

Tight variable domains may accelerate the solving: see the next slides for how to do that.

Literals

The following literals (or: constants) can be used:

- Boolean: `true` and `false`
- Integers: in decimal, hexadecimal, or octal format
- Sets: between curly braces, for example `{1,3,5}`, or as integer intervals, for example `10..30`
- 1d arrays: between square brackets, say `[6,3,1,7]`
- 2d arrays: A vertical bar `|` is used before the first row, between rows, and after the last row; for example `[|11,12,13,14|21,22,23,24|31,32,33,34|]`
- For higher-dimensional arrays, see slide 11

Careful: The indices of arrays start from `1` by default.

Declarations of Parameters and Variables

```
int: n = 4;
par int: p;
p = 10;
set of int: Primes = {2,3,5,7,11,13};
var int: x;
var 0..23: hour = x + n;
var set of Primes: Taken;
```

- A parameter must be instantiated, once, to a literal; its declaration can be separated from its instantiation in the model (`p`), a datafile, the command line, or the IDE.
- A variable can be constrained at its declaration (`hour`).
- The domain of a decision variable can be tightened by replacing its type by a set of values of that type:
  - `x` must take an integer value.
  - `hour` must take an integer value between `0` and `23`.
  - `Taken` must be a subset of `{2,3,5,7,11,13}`.

Array and Set Comprehensions

An array or set can be built by a comprehension, using the notation $[\sigma \mid \gamma]$ or $\{\sigma \mid \gamma\}$, where σ is an expression evaluated for each element generated by the generator γ: a generator introduces one or more identifiers with values drawn from integer sets, optionally followed by a test.

Examples

```
[x * 2 | x in 1..8]
      evaluates to [2,4,6,8,10,12,14,16]
[x * y | x, y in 1..3 where x < y]
      evaluates to [2,3,6]
[x + 2*y | x in 1..3, y in 1..4]
      evaluates to [3,5,7,9,4,6,8,10,5,7,9,11]
{x + 2*y | x in 1..3, y in 1..4}
      evaluates to {3,4,5,6,7,8,9,10,11}
```

Indexing: Syntactic Sugar

For example,

```
sum(i,j in 1..5)(i*j)
```

is syntactic sugar for

```
sum([i*j | i, j in 1..5])
```

This works for any function or predicate that takes an array as unique argument. In particular:

```
forall(i in 1..9)(x[i+1] = x[i] + y[i]);
```

is syntactic sugar for

```
forall([x[i+1] = x[i] + y[i] | i in 1..9]);
```

where `forall(array[int] of var bool: B)` is a function that returns the conjunction of all expressions in `B`: it generalises the 2-ary logical-and connective (`/\`).

Array Manipulation

- Changing the number of dimensions and their ranges, provided the numbers of elements match:

  `array1d(5..10,[|3,2|5,4|6,1|])`

  `array2d(1..2,1..3,[2,7,3,7,4,9])`

  and so on, until `array6d`.

  Tip: Try and keep your ranges starting from `1`:

  - It is easier to read a model under this usual convention.
  - Subtle errors may occur otherwise.

- Concatenation: for example, `[1,2] ++ [3,4]`.

Subtyping

A parameter can be used wherever a variable is expected. This extends to arrays: for example, a predicate or function expecting an argument of type `array[int] of var int` can be passed an argument of type `array[int] of int`.

The types `bool` and `int` are disjoint, but one can coerce from `bool` to `int` using the `bool2int` function: `bool2int(false) = 0` and `bool2int(true) = 1`. This coercion is automatic but should be explicit for clarity.

In mathematics we use the Iverson bracket for this purpose: we define $[\phi] = 1$ iff formula ϕ is true, and $[\phi] = 0$ otherwise.

Option Variables

An option variable is a decision variable that can also take the special value $\perp$, to be read “bottom”.

A variable is declared optional with the keyword `opt`.

For example, `var opt 1..4: x` declares a variable `x` of domain `{1,2,3,4,⊥}`.

We will not cover the use of option variables in this course. However, one can see them:

- In the documentation:
  `var int` is a subtype of `var opt int`.
- In error messages:
  This is probably a sign that a model is too complicated.

Constraints

A constraint is the keyword `constraint` followed by a Boolean expression that must be true in every solution.

Examples

```
constraint x < y;

constraint sum(Q) = 0 /\ alldifferent(Q);

constraint if x < y then x = y else x > y endif;
```

Constraints separated by a semi-colon (`;`) are implicitly connected by the 2-ary logical-and connective (`/\`).

What does `constraint x = x + 1` mean?
MiniZinc is declarative and has no destructive assignment: this equality constraint is not satisfied by any value for `x`.

Objective

The `solve` item gives the objective of the problem:

- `solve satisfy;`
  The objective is to solve a satisfaction problem.
- `solve minimize x;`
  The objective is to minimise the value of variable `x`.
- `solve maximize abs(x) * y;`
  The objective is to maximise the value of the expression `abs(x) * y`.

MiniZinc does not support multi-objective optimisation yet: multiple objective functions must either be aggregated into a weighted sum, or be handled outside a MiniZinc model.

Output Command

The `output` item prescribes what to print upon finding a solution: the keyword `output` is followed by a string array.

```
output [show(X)];
```

The function `show` returns a string representing the value of its argument expression.

```
output ["solution:"] ++ [if X[i]>0 then show(2*X[i])
  ++ ", " else " , " endif | i in 1..10];
```

The operator `++` concatenates two strings or two arrays.

`"X = \(X),"` is equivalent to `"X = "++show(X)++","`.

Tests

Conditional expressions can be formulated as follows:

- Conditional: `if` θ `then` ϕ_1 `else` ϕ_2 `endif`
- Comprehension: `[i | i in` ρ `where` θ`]`

The expressions ϕ_1 and ϕ_2 must have the same type.

Strong warning: The test θ after `if` or `where` can depend on variables: with great power comes great responsibility!

For example:

```
enum N; set of int: M; array[N] of var M: X;
array[S] of var M: Y = [X[i] | i in N where X[i]>0];
constraint sum(Y) < 7;
```

This yields an error message with `var opt` (see slide 13) as the indices `S` cannot be determined when flattening and cannot just be set to `N`. But the following variant works:

```
constraint sum([X[i] | i in N where X[i]>0]) < 7;
```

Operators and Functions

- Booleans: `not`, `/\`, `\/`, `<->`, `->`, `<-`, `xor`, `forall`, `exists`, `xorall`, `iffall`, `clause`, `bool2int`
  Avoid arbitrarily nested Boolean expressions, such as `forall...exists...forall...`!
- Integers: `+`, `-`, `*`, `div`, `mod`, `abs`, `pow`, `min`, `max`, `sum`, `product`, `=`, `<`, `<=`, `=>`, `>`, `!=`, `..`
  Avoid `div`, `mod`, and `pow` on variables, if possible!
- Sets: `union`, `intersect`, `diff`, `symdiff`, `card`, `in`, `subset`, `superset`, `set2array`, `array_union`, `array_intersect`
- Strings: `++`, `concat`, `join`
- Arrays: `length`, `index_set`, `index_set_1of2`, `index_set_2of2`, ..., `index_set_6of6`, `array1d`, `array2d`, ..., `array6d`

Predicates and Functions

MiniZinc offers a large collection of predefined predicates and functions to enable a medium level at which models can be formulated: see Topic 3: Constraint Predicates.

Each predefined constrained function is defined by the use of the corresponding constraint predicate, possibly upon introducing a new variable.

Example

`count(A,v)>m` is replaced by `count(A,v,c) /\ c>m`.

It is also possible for modellers to define their own functions and predicates, as discussed at slide 24.

UPPSALA UNIVERSITET

Reification

Reification enables the reasoning about the truth of a constraint or a Boolean expression.

Example

```
constraint x < y;
```

requires that `x` be smaller than `y`.

```
constraint b <-> x < y;
```

requires that the Boolean variable `b` be `true` if and only if (iff) `x` is smaller than `y`: the constraint `x < y` is said to be reified, and `b` is called its reification.

Reification is a powerful mechanism that enables:

- higher-level modelling;
- easier implementation of the logical connectives.

The expression `bool2int(`γ`)`, for a constraint or Boolean expression γ, is an integer expression, with the truth of γ represented by `1` and its falsity by `0`.

Example (Cardinality constraint)

Constrain one or two of three constraints γ_1, γ_2, γ_3 to hold:
`bool2int(`γ_1`)+bool2int(`γ_2`)+bool2int(`γ_3`) in {1,2}`

Reification comes with some drawbacks:

- Inference and relaxation may be poor: slow solving.
- Not all constraints can be reified in MiniZinc.

Example (Soft Constraints: Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to be next to person `Pref[k,2]`. Maximise the number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(           bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

Example (Soft Constraints: Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to be next to person `Pref[k,2]`. Maximise the number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(         bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

UPPSALA UNIVERSITET

The MiniZinc Language

Modelling

Set Variables &Constraints

Example (Soft Constraints: Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to be next to person `Pref[k,2]`. Maximise the number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(           bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

Example (Soft Constraints: Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to be next to person `Pref[k,2]`. Maximise the number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(          bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

Example (Soft Constraints: Weighted Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
array[1..q] of int: Weight;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to pay `Weight[k]` to be next to person `Pref[k,2]`. Maximise the weighted number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(          bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

Example (Soft Constraints: Weighted Photo Problem)

An enumeration `Persons` of `n` people lines up for a photo.

```
array[1..q,1..2] of Persons: Pref;
array[1..q] of int: Weight;
```

Preference `k` in `1..q` denotes that person `Pref[k,1]` wants to pay `Weight[k]` to be next to person `Pref[k,2]`. Maximise the weighted number of satisfied preferences.

Let decision variable `Pos[p]` denote the position in `1..n`, in left-to-right order, of person `p` in `Persons` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint alldifferent(Pos);
```

The objective is:

```
solve maximize sum(k in 1..q)
(Weight[k]*bool2int(abs(Pos[Pref[k,1]]-Pos[Pref[k,2]])=1));
```

Example (Sum of reified constraints)

The expression `sum(i in 1..n)(bool2int(A[i]=v))` denotes the number of elements of array `A` that equal `v`.

This idiom is very common in constraint-based models. So:

Example (The `count` constraint predicate)

The constraint `count(A,v,c)` holds iff variable `c` has the number of variables of array `A` that equal variable `v`.

For other predicates, see Topic 3: Constraint Predicates.

Example (The `count` constrained function)

The expression `count(A,v)` denotes the number of variables of array `A` that equal variable `v`.

Functional constraint predicates are available as functions.

Predicate and Function Definitions

Examples

```
function int: double(int: x);
function var int: double(var int: x);

predicate pos(var int: x);
function var bool: neg(var int: x);
```

A predicate can be used as a function returning `var bool`. For example, `bool2int(pos(a))` can be used.

Function and predicate names can be overloaded.

The body of a predicate or function definition is an expression of the same type as the returned value.

Examples

```
function int: double(int: x) = 2 * x;
function var int: double(var int: x) = 2*x;

predicate pos(var int: x) = x > 0;
function var bool: neg(var int: x) = x < 0;
```

One can use `if ... then ... else ... endif`, predicates and functions, such as `forall` and `exists`, as well as `let` expressions (see next slide) in the body of a predicate or function definition.

Let Expressions

One can introduce local identifiers with a let expression.

Examples

```
function int: double(int: x) =
  let { int: y = 2 * x } in y;

function var int: double(var int: x) =
  let { var int: y = 2 * x } in y;

function var int: double(var int: x) =
  let { var int: y;
        constraint y = 2 * x
  } in y;
```

The 2nd and 3rd functions are equivalent:
each use adds a decision variable to the model.

Constraints in Let Expressions

What is the difference between the next two definitions?

```
predicate posProd(var int: x, var int: y) =
  let { var int: z; constraint z = x * y
  } in z > 0;

predicate posProd(var int: x, var int: y) =
  let { var int: z
  } in z = x * y /\ z > 0;
```

Their behaviour is different in a negative context, such as `not posProd(a,b)`:

- The 1st one then ensures `a * b = z /\ z <= 0`.
- The 2nd one then ensures `a * b != z \/ z <= 0` and leaves `a` and `b` unconstrained.

Using Predicates and Functions

Advantages of using predicates and functions in a model:

- Software engineering good practice:
  - Reusability
  - Readability
  - Modularity
- The model might be solved more efficiently:
  - Better common-subexpression elimination.
  - The definitions can be technology- or solver-specific. If a predefined constraint predicate is a built-in of a solver, then its solver-specific definition is empty!

Remarks

- The order of model items does not matter.
- One can include other files.
  Example: `include "globals.mzn"`.
- The following functions are useful for debugging:
  - `assert(`θ`,"error message")`
    If the Boolean expression θ evaluates to `false`, then abort with the error message, otherwise return `true`.
  - `trace("message",` ϕ`)`
    Print the message and return the expression ϕ.
  - …

Other Modelling Languages

- OPL: https://www.ibm.com/analytics/optimization-modeling-interfaces
- Comet: https://mitpress.mit.edu/books/constraint-based-local-search
- Essence and Essence′: https://constraintmodelling.org
- Zinc: https://dx.doi.org/10.1007/s10601-008-9041-4
- AIMMS: https://aimms.com
- AMPL: https://ampl.com
- FICO Xpress Insight: https://www.fico.com/en/products/fico-xpress-insight
- GAMS: https://gams.com
- SMT-lib: http://www.smt-lib.org
- …

Outline

From a Problem to a Model

What is a good model for a constraint problem?

- A model that correctly represents the problem
- A model that is easy to understand and maintain
- A model that is solved efficiently, that is:
  - short solving time to find one, all, or best solution(s)
  - good solution within a limited amount of time
  - small search space (under constructive search)

Food for thought: What is correct, easy, short, good, . . . ?

Modelling Issues

Modelling is still more an Art than a Science:

- Choice of the decision variables and their domains
- Choice of the constraint predicates, in order to model the objective function, if any, and the constraints
- Optional for CP and LCG:
  - Choice of the consistency for each constraint
  - Choice of the variable selection strategy for search
  - Choice of the value selection strategy for search

  See Topic 8: Inference & Search in CP & LCG.

Make the model correct before making it efficient!

Choice of the Decision Variables

Examples (Alphametic Problems)

SEND + MORE = MONEY:

Model without carry variables: 19 of 23 CP nodes visited:

$$1000 \cdot (S + M) + 100 \cdot (E + O) + 10 \cdot (N + R) + (D + E)$$
$$= 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y$$

Model with carry variables: 23 of 29 CP nodes are visited:

$$D + E = 10 \cdot C_1 + Y \;\wedge\; N + R + C_1 = 10 \cdot C_2 + E$$
$$\wedge\; E + O + C_2 = 10 \cdot C_3 + N \;\wedge\; S + M + C_3 = 10 \cdot M + O$$

GERALD + DONALD = ROBERT:

The model with carry variables is more effective in CP: only 791 of 869 nodes are visited, rather than 13,795 of 16,651 search nodes for the model without carry variables.

Choice of the Constraint Predicates

Example (The `alldifferent` constraint predicate)

The constraint `alldifferent([A[i] | i in 1..n])` usually leads to faster solving than its definition by a conjunction of $\Theta(\mathtt{n}^2)$ disequality constraints:

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

For more examples, see Topic 3: Constraint Predicates.

Guidelines: Reveal Problem Structure

- Use few decision variables.
- Give tight domains to the decision variables.
- Avoid division and power constraints (`div`, `mod`, `pow`).
- Avoid the disjunction of constraints (`\/`, `<-`, `->`, `<->`).
- Express the problem concisely (Topic 3: Constraint Predicates).
- Precompute solutions to a sub-problem into a table (Topic 3: Constraint Predicates, Topic 4: Modelling).
- Use implied constraints (Topic 4: Modelling).
- Use different viewpoints (Topic 4: Modelling).
- Exploit symmetries (Topic 5: Symmetry).

Careful: These guidelines of course have their exceptions!
It is important to test empirically several combinations of model, solver, and solving technology.

Use Few Decision Variables

When appropriate, use a single integer variable instead of an array of Boolean variables:

Example

Assume Joe must be assigned to exactly one task in `1..n`:

- Use a single integer variable, `var 1..n: joesTask`, representing *which* task Joe is assigned to.
- Don't use `array[1..n] of var bool:joesTask`, each element `joesTask[t]` representing *whether* (`true`) or not (`false`) Joe is assigned to task `t`, plus `count(joesTask,true) = 1`.

When appropriate, use a single set variable instead of an array of Boolean or integer variables: see slides 46 and 48.

Give Tight Domains to the Variables

Without doing all the work of the solver, manually tightening the domains of the variables may accelerate the solving. In particular, try and avoid `var int`:

Example

If the integer variable `s` represents the starting time of some task, then declare its domain with `var 0..horizon: s`, where `horizon` is suitably large, rather than `var int: s`.

Domain information may be exploited during flattening, so avoid setting a domain by constraints:

Counterexample

Do not reformulate `var 0..horizon: s` as follows:

```
var int: s; constraint 0<=s /\ s<=horizon;
```

Avoid Division and Power Constraints

The use of `div` and `mod` often slows the solving down. Try and use $\star$ or `table` (see Topic 3: Constraint Predicates).

Example

```
constraint q = x div k  /\ r = x mod k;
```

over variables `x`, `q`, `r` and parameter `k` is equivalent to

```
constraint x = q * k + r; var 0..k-1: r;
```

Similarly,

```
solve minimize sum(X) div n; % minimise the average
```

over `n` variables `X[i]` and parameter `n` is equivalent to

```
solve minimize sum(X);         % minimise the sum
output [show(sum(X) div n)]; % output the average
```

Try and avoid constraints containing `pow(x,y)` for x^y.

Avoid the Disjunction of Constraints

The disjunctive combination of constraints (with `\/`, `<-`, `->`, or `<->`) often makes the solving slow. Try and express disjunctive combinations of constraints otherwise.

Example

```
constraint x = 0 \/ x = 9;
```

is logically equivalent to

```
constraint x in {0,9};
```

and, even better, to

```
var {0,9}: x;
```

Disjunction or other sources of slow solving may also be introduced by `not`, so try and avoid negation as well.

Example

```
constraint b -> x = 9;
constraint (not b) -> x = 0;
```

is logically equivalent to (recall that `bool2int(true)=1`)

```
constraint x = 9 * bool2int(b);
```

and to (note that array indexing starts at 1)

```
constraint x = [0,9][1+bool2int(b)];
```

But beware of such premature fine-tuning of a model!

The following versions are clearer and often good enough:

```
constraint x = if b then 9 else 0 endif;
```

and

```
constraint if b then x=9 else x=0 endif;
```

Express the Problem Concisely

Whenever possible, use a single predefined constraint predicate instead of a long-winded formulation.

Example (The `alldifferent` constraint predicate)

The constraint `alldifferent([A[i] | i in 1..n])` usually leads to faster solving than its definition by a conjunction of $\Theta(\mathtt{n}^2)$ disequality constraints:

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

For more examples, see Topic 3: Constraint Predicates.

Outline

1. The MiniZinc Language

2. Modelling

3. Set Variables &Constraints

UPPSALA UNIVERSITET

Motivating Example 1

Example (Agricultural experiment design, AED)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley							
corn							
millet							
oats							
rye							
spelt							
wheat							

Constraints to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

Instance: 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.

Motivating Example 1

Example (Agricultural experiment design, AED)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	✓	–	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

Constraints to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

Instance: 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.

Example (BIBD *integer* model: ✓ ⇝ 1 and – ⇝ 0)

```
int: nbrBlocks; int: nbrVarieties;
set of int: Blocks = 1..nbrBlocks;
set of int: Varieties = 1..nbrVarieties;
int: blockSize; int: sampleSize; int: balance;
array[Varieties,Blocks] of var 0..1: BIBD;
solve satisfy;
constraint forall(b in Blocks)
(blockSize = sum(BIBD[..,b]));
constraint forall(v in Varieties)
(sampleSize = sum(BIBD[v,..]));
constraint forall(v, w in Varieties where v < w)
(balance = sum(b in Blocks)(BIBD[v,b]*BIBD[e,b]));
```

At Topic 1: Introduction, we used `count` instead of `sum`.

Example (Instance data for our AED)

```
nbrBlocks = 7; nbrVarieties = 7;
blockSize = 3; sampleSize = 3; balance = 1;
```

UPPSALA UNIVERSITET

The MiniZinc Language

Modelling

Set Variables &Constraints

Example (Idea for another BIBD model)

barley	{plot1,	plot2,	plot3				}
corn	{plot1,			plot4,	plot5		}
millet	{plot1,					plot6,	plot7}
oats	{	plot2,		plot4,		plot6	}
rye	{	plot2,			plot5,		plot7}
spelt	{		plot3,	plot4,			plot7}
wheat	{		plot3,		plot5,	plot6	}

Constraints to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

UPPSALA UNIVERSITET

The MiniZinc Language

Modelling

Set Variables &Constraints

Example (BIBD *set* model: a block *set* per variety)

```
...
...
...
...
array[Varieties] of var set of Blocks: BIBD;
...
...
(blockSize =
  sum(v in Varieties)(bool2int(b in BIBD[v])));
...
(sampleSize = card(BIBD[v]));
...
(balance = card(BIBD[v] intersect BIBD[w]));
```

At Topic 1: Introduction, `bool2int` was not covered yet.

Example (Instance data for our AED)

```
nbrBlocks = 7; nbrVarieties = 7;
blockSize = 3; sampleSize = 3; balance = 1;
```

Motivating Example 2[1]

Example (Hamming code: problem)

The Hamming distance between two same-length strings is the number of positions at which the two strings differ. Examples: $h(10001, 01001) = 2$ and $h(11010, 11110) = 1$.

ASCII has codewords of $m = 8$ bits for $n = 2^m$ symbols, but the least Hamming distance is $d = 1$: no robustness!

Toward high robustness in data transmission, we want to generate a codeword of m bits for each of the n symbols of an alphabet, such that the Hamming distance between any two codewords is at least some given constant d.

[1] Based on material by Christian Schulte

Example (Hamming code: model)

We encode a codeword of m bits as the set of positions of its unit bits, the least significant bit being at position 1. Example: 10001 is encoded as $\{1, 5\}$, and 01001 as $\{1, 4\}$. In general: $b_m \cdots b_1$ is encoded as $\{1 \cdot b_1, \ldots, m \cdot b_m\} \setminus \{0\}$. So the Hamming distance between two codewords is $u - i$, where u is the size of the union of their encodings and i is the size of the intersection of their encodings, that is the size of the symmetric difference of their encodings. Hence:

```
array[1..n] of var set of 1..m: C;
constraint forall(i, j in 1..n where i < j)
  (card(C[i] symdiff C[j]) >= d);
```

Definition

A set (decision) variable takes a set as value, and has a set of sets as domain. For its domain to be finite, a set variable must be a subset of a finite set.

Set-constraint predicates exist for the following semantics:

- Cardinality: $|S| = n$
- Membership: $n \in S$
- Equality: $S_1 = S_2$
- Disequality $S_1 \neq S_2$
- Subset: $S_1 \subseteq S_2$
- Union: $S_1 \cup S_2 = S_3$
- Intersection: $S_1 \cap S_2 = S_3$
- Difference: $S_1 \setminus S_2 = S_3$
- Symmetric difference: $(S_1 \cup S_2) \setminus (S_1 \cap S_2) = S_3$
- Order: $S_1 \subseteq S_2 \vee \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$
- Strict order: $S_1 \subset S_2 \vee \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$

where the S_i are set variables and n is an integer variable.
Avoid set variables in M4CO: few solvers support them well.