Topic 2: Basic Modelling
(Version of 5th August 2022)

Pierre Flener and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL442:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

1Many thanks to Guido Tack for feedback
Outline

1. The MiniZinc Language

2. Modelling

3. Set Variables & Constraints

4. Modelling Checklist
Outline

1. The MiniZinc Language

2. Modelling

3. Set Variables & Constraints

4. Modelling Checklist
MiniZinc Model

A MiniZinc model may comprise the following items:

- Parameter declarations
- Variable declarations
- Predicate and function definitions
- Constraints
- Objective
- Output
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** \( \tau \): set of elements of type \( \tau \), which is int, bool, enum, float, or string
- **array** \([\rho]\) **of** \( \tau \): possibly multidimensional array of elements of type \( \tau \), which is not an array; each index range in \( \rho \) is an enumeration or an integer range \( \alpha .. \beta \)

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 feasible (and 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 domains for variables might accelerate the solving: see the next slides for how to do that.
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 ranges, 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

1 int: n = 4;
2 par int: p;
3 p = 10;
4 var 0..23: hour;
5 set of int: Primes = {2,3,5,7,11,13};
6 var set of Primes: Taken;
7 var int: nbr = card(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.

- The domain of a decision variable is tightened by replacing its type by a set of values of that type:
  - \texttt{hour} must take an integer value from 0 to 23
  - \texttt{Taken} must be a subset of \{2, 3, 5, 7, 11, 13\}
  - \texttt{nbr} is equality-constrained at its declaration: its domain is inferred to be 0..6 (see slide 39 for more)
Array and Set Comprehensions

An array or set can be built by a comprehension, using the notation `[σ | γ]` or `{σ | γ}`, 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 under a `where` test.

Examples

1. `[i*2 | i in 1..8]` evaluates to `[2,4,6,8,10,12,14,16]`
2. `[i*j | i,j in 1..3 where i<j] % both i and j in 1..3` evaluates to `[2,3,6]`
3. `[i + 2*j | i in 1..3, j in 1..4]` evaluates to `[3,5,7,9,4,6,8,10,5,7,9,11]`
4. `{i + 2*j | i in 1..3, j in 1..4}` evaluates to `{3,4,5,6,7,8,9,10,11}`
5. `Sudoku[row,..] % slicing` is syntactic sugar for `[Sudoku[row,col] | col in 1..9]`
Indexing: Syntactic Sugar

For example,

\[
\text{sum}(i, j \text{ in } 1..n \text{ where } i<j)(X[i]*X[j])
\]

is syntactic sugar for

\[
\text{sum}([X[i]*X[j] \mid i, j \text{ in } 1..n \text{ where } i<j])
\]

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

\[
\text{forall}(i \text{ in } 1..n)(Z[i] = X[i] + Y[i]);
\]

is syntactic sugar for

\[
\text{forall}([Z[i] = X[i] + Y[i] \mid i \text{ in } 1..n]);
\]

where a \text{forall}(\text{array[int] of var bool: } B) constraint holds if and only if (iff) all the expressions in $B$ hold: it generalises the 2-ary logical-and connective ($\land$).
Array Manipulation

- Changing the number of dimensions and their index ranges, provided the numbers of elements match:
  
  \[
  \text{array1d}(5..10, [3, 2, 5, 4, 6, 1])
  \]
  
  \[
  \text{array2d}(1..2, 1..3, [2, 7, 3, 7, 4, 9])
  \]
  
  and so on, until \text{array6d}.

Try and keep your index ranges starting from 1:

- It is easier to read a model under this usual convention.
- Subtle errors might 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 type `bool` is a subtype of the type `int`. One can coerce from `bool` to `int` using the `bool2int` function:

\[
\text{bool2int}(\text{true}) = 1 \quad \text{and} \quad \text{bool2int}(\text{false}) = 0.
\]

This coercion is automatic when needed.

In mathematics we use the Iverson bracket for this purpose: we define \([\psi] = 1\) iff formula \(\psi\) is true, and \([\psi] = 0\) otherwise.
Option Variables

An option variable is a decision variable that can also take the special value $<>$ indicating the absence of the variable.

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, <>}`.

Do not use `explicit` option variables in this course. However, one can see them:

- In the documentation: for example, `var int` is a subtype of `var opt int`.
- In error messages, due to `implicit` option variables being made explicit while flattening, but things getting too complex: see the symptomatic example at slide 21.
Constraints

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

### Examples

```plaintext
constraint x < y;
constraint sum(Q) = 0 \land all_different(Q);
```

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

**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`. MiniZinc allows the syntax `constraint x == x + 1`, but note that MiniZinc is syntax for mathematics and logic!
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 objective function `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

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..n];
```

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

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

The search strategy of the CP backend Gecode depends on the decision variables mentioned in the `output` statement.
Operators and Functions

- **Booleans:** `not`, `\`, `\`, `<-->`, `->`, `<-`, `xor`, `forall`, `exists`, `xorall`, `iffall`, `clause`, `bool2int`
  Beware of arbitrarily nested logical quantifications, such as `forall(...exists(...forall(...)))`!

- **Integers:** `+`, `-`, `*`, `div` (`/ is for float`), `mod`, `abs`, `pow`, `min`, `max`, `sum`, `product`, `= (or ==)`, `<`, `<=`, `=>`, `>`, `!=`
  Beware of `div`, `mod`, and `pow` on variables!

- **Sets:** `..`, `in`, `card`, `subset`, `superset`, `union`, `array_union`, `intersect`, `array_intersect`, `diff`, `symdiff`, `set2array`

- **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 high 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

\[
\text{count}(A, v) > m \text{ is defined by count}(A, v, c) \land c > m.
\]

It is also possible for modellers to define their own functions and predicates, as discussed at slide 25.
Reification

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

Example

```plaintext
constraint x < y;
```

requires that \( x \) be smaller than \( y \).

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

requires that the Boolean variable \( b \) take the value \texttt{true} if and only if \( x \) 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 \( \text{bool2int}(\phi) \), for a Boolean expression \( \phi \), denotes the integer 1 if \( \phi \) is true, and 0 if \( \phi \) is false.

**Example (Cardinality constraint)**

Constrain one or two of three constraints \( \gamma_1, \gamma_2, \gamma_3 \) to hold:

\[
\text{bool2int}(\gamma_1) + \text{bool2int}(\gamma_2) + \text{bool2int}(\gamma_3) \in \{1,2\}
\]

As \( \text{bool2int} \) coercion is automatic, one can actually write:

\[
\gamma_1 + \gamma_2 + \gamma_3 \in \{1,2\}
\]

However, as a coding convention, we recommend to write:

\[
(\gamma_1) + (\gamma_2) + (\gamma_3) \in \{1,2\}
\]

thereby mimicking the Iverson bracket (see slide 12).

Reification (implicit via \( \text{bool2int} \) and \( (\ldots) \)) has pitfalls:

- Inference and relaxation might be poor: slow solving.
- Not all constraints can be reified in MiniZinc, such as some of those in Topic 3: Constraint Predicates.
A conditional expression can be formulated as follows:
- **Conditional:** if \( \theta \) then \( \phi_1 \) else \( \phi_2 \) endif
- **Comprehension:** \([i \mid i \in \sigma \text{ where } \theta]\)

The expressions \( \phi_1 \) and \( \phi_2 \) must have the same type.

The test \( \theta \) after if or where may depend on variables, but this can be a source of inefficiency, unexpected behaviour (see documentation Section 2.4.2), or impossible flattening!

**Example**

```plaintext
1 enum I; set of int: T; array[I] of var T: X;
2 array[I] of var T: Y = [X[i] | i in I where X[i]>0];
  constraint sum(Y) < 7;
```

This yields an error message with var opt (see slide 13) as the indices of \( Y \) cannot be determined when flattening and cannot just be set to \( I \). But the following works:

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

and so does the use of implicit reification, possibly better:

```plaintext
2 constraint sum([[(X[i]>0) * X[i] | i in I]]) < 7;
```
Example (Soft Constraints: Photo Problem)

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

```mini
enum PrefRoles = {who, whom}; % do not use 1..2
array[1..q, PrefRoles] of Persons: Pref;
```

Preference $k$ in $1..q$ denotes that person $\text{Pref}[k, \text{who}]$ wants to be next to $\text{Pref}[k, \text{whom}]$.

Maximise the number of satisfied preferences.

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

The array $\text{Pos}$ must form a permutation of the positions:

```mini
constraint all_different(Pos);
```

The objective, formulated using implicit reification, is:

```mini
solve maximize sum(k in 1..q)
( abs(\text{Pos}[\text{Pref}[k, \text{who}]] - \text{Pos}[\text{Pref}[k, \text{whom}]]) = 1);
```
Example (Soft Constraints: Weighted Photo Problem)

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

```mini
enum PrefRoles = {who, whom}; % do not use 1..2
array[1..q, PrefRoles] of Persons: Pref;
array[1..q] of int: Fee;
```

Preference $k$ in $1..q$ denotes that person $\text{Pref}[k, \text{who}]$ wants to pay $\text{Fee}[k]$ to be next to $\text{Pref}[k, \text{whom}]$.

Maximise the weighted number of satisfied preferences.

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

The array $\text{Pos}$ must form a permutation of the positions:

```mini
constraint all_different(\text{Pos});
```

The objective, formulated using implicit reification, is:

```mini
solve maximize sum($k$ in 1..$q$)
( abs(\text{Pos}[\text{Pref}[k, \text{who}]]\text{-}\text{Pos}[\text{Pref}[k, \text{whom}]] )=1 );
```
Example (Soft Constraints: Weighted Photo Problem)

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

```plaintext
enum PrefRoles = {who, whom}; % do not use 1..2
array[1..q, PrefRoles] of Persons: Pref;
array[1..q] of int: Fee;
```

Preference \( k \) in 1..\( q \) denotes that person \( \text{Pref}[k, \text{who}] \) wants to pay \( \text{Fee}[k] \) to be next to \( \text{Pref}[k, \text{whom}] \).

Maximise the weighted number of satisfied preferences.

Let decision variable \( \text{Pos}[p] \) denote the position in 1..\( n \), in left-to-right order, of person \( p \) in Persons on the photo.

The array \( \text{Pos} \) must form a permutation of the positions:

```plaintext
constraint all_different(Pos);
```

The objective, formulated using implicit reification, is:

```plaintext
solve maximize sum(k in 1..q) (Fee[k] * (abs(Pos[Pref[k, who]] - Pos[Pref[k, whom]]) = 1));
```
Example (Sum of unweighted reified constraints)
The expression $\text{sum}(i \text{ in } \text{index_set}(A)) \ (A[i]=v)$ denotes the count of occurrences in array $A$ of $v$.

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

Definition (The \textit{count} constraint predicate)
The constraint $\text{count}(A, v, c)$ holds iff variable $c$ has the number of variables of array $A$ that are equal to variable $v$.

For other predicates, see Topic 3: Constraint Predicates.

Definition (The \textit{count} constrained function)
The expression $\text{count}(A, v)$ denotes the number of variables of array $A$ that are equal to variable $v$.

Example (Unweighted Photo Problem, revisited)
\begin{verbatim}
solve maximize \text{count}([\text{abs}(\text{Pos[Pref[k,who]]-Pos[Pref[k,whom]])} \mid k \text{ in } 1..q], 1);
\end{verbatim}

Functional constraint predicates are available as functions.
Predicate and Function Definitions

Examples

1. `function int: double(int: x);`
2. `function var int: double(var int: x);`
3. `predicate pos(var int: x);`
4. `function var bool: neg(var int: x);`

A predicate is a function denoting a `var bool`:

Examples

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

Function and predicate names can be overloaded.
The body of a predicate or function definition is an expression of the same type as the denoted value.

### Examples

1. `function int: double(int: x) = 2*x;`
2. `function var int: double(var int: x) = 2*x;`
3. `predicate pos(var int: x) = x > 0;`
4. `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 the next slide) in the body of a predicate or function definition.
Let Expressions

One can introduce local identifiers with a let expression.

Examples

```mini
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 second and third 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?

1. \[
\text{predicate posProd(var int: } x, \text{ var int: } y) = \\
\text{ let } \{ \text{ var int: } z; \text{ constraint } z = x \times y \} \text{ in } z > 0;
\]

2. \[
\text{predicate posProd(var int: } x, \text{ var int: } y) = \\
\text{ let } \{ \text{ var int: } z \} \text{ in } z = x \times y \land z > 0;
\]

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

- The 1st one then ensures \(a \times b = z \land z \leq 0\).
- The 2nd one then ensures \(a \times b \neq z \lor z \leq 0\) 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 identity!
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 $\theta$ evaluates to `false`, then abort with the error message, otherwise denote `true`.
  
  - `trace("message", ϕ)`
    Print the message and denote the expression $\phi$.
  
  - ...
Other Modelling Languages

- **OPL**: [https://www.ibm.com/analytics/optimization-modeling-interfaces](https://www.ibm.com/analytics/optimization-modeling-interfaces)
- **Comet**: [https://mitpress.mit.edu/books/constraint-based-local-search](https://mitpress.mit.edu/books/constraint-based-local-search)
- **Essence and Essence’**: [https://constraintmodelling.org](https://constraintmodelling.org)
- **Zinc**: [https://dx.doi.org/10.1007/s10601-008-9041-4](https://dx.doi.org/10.1007/s10601-008-9041-4)
- **AIMMS**: [https://aimms.com](https://aimms.com)
- **AMPL**: [https://ampl.com](https://ampl.com)
- **GAMS**: [https://gams.com](https://gams.com)
- **SMT-lib**: [https://smtlib.cs.uiowa.edu](https://smtlib.cs.uiowa.edu)
- **…**
Outline

1. The MiniZinc Language
2. Modelling
3. Set Variables & Constraints
4. Modelling Checklist
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 systematic 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 a 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:

\[
D + E = 10 \cdot C_1 + Y \land N + R + C_1 = 10 \cdot C_2 + E \\
\land E + O + C_2 = 10 \cdot C_3 + N \land S + M + C_3 = 10 \cdot M + O
\]

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 `all_different` constraint predicate)

The constraint `all_different(A)` for an array `A` of size `n` usually leads to faster solving than its definition by a conjunction of \( \frac{n(n-1)}{2} \) disequality constraints:

\[
\forall (i, j \text{ in } \text{index}_\text{set}(A) \text{ where } i < j) (A[i] \neq A[j])
\]

For more examples, see Topic 3: Constraint Predicates.
Guidelines: Reveal Problem Structure

- Use few decision variables, and declare tight domains
- Beware of nonlinear and power constraints: pow
- Beware of division constraints: div and mod (avoid /)
- Beware of disjunction & negation: \/, -, ->, <->, not
- 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 49 & 51.
Declare the Variables with Tight Domains

Tight domains for variables might accelerate the solving.
Beware of `var int` for non equality-constrained variables.

Example (Use parameters for declaring tight domains)

If the variable `t` denotes a time, then write `var 0..h: t`, where horizon `h` is a parameter, instead of `var int: t`.

Example (Derive tight domains from the parameters)

```plaintext
1 int: p; int:c=ceil(pow(p,1/3)); int:s=ceil(sqrt(p));
2 var 1..c: x; var 1..s: y; var 1..p: z; % no "var int"
3 constraint x * y * z = p \&\& x <= y \&\& y <= z;
```
Beware of Nonlinear and Power Constraints

Constraining the product of two or more variables often makes the solving slow. Try and find a linear reformulation.

Example

The model snippet

```plaintext
array[1..n] of var 0..1: X;
array[1..n] of var 0..1: Y;
constraint count([X[i]*Y[i] | i in 1..n], 1) = b;
```

should be reformulated as:

```plaintext
array[1..n] of var 0..1: X;
array[1..n] of var 0..1: Y;
constraint count([X[i]+Y[i] | i in 1..n], 2) = b;
```
Beware of Division Constraints

The use of `div` and `mod` often makes the solving slow. Use `table` (see Topic 3: Constraint Predicates) or reformulate.

Example

The model snippet

```plaintext
solve minimize sum(X) div n;  % minimise the average over n variables X[i] and parameter n should become:
```

```plaintext
solve minimize sum(X);  % minimise the sum
output [show(sum(X) div n)];  % output the average
```
Beware of Disjunction and Negation

The disjunction of constraints (with \(\lor\), xor, \(<-\), \(<->\), exists, xorall, if \(\theta\) then \(\phi\) else \(\psi\) endif) often makes the flat code long and the solving slow. Try and express disjunctive combinations of constraints otherwise.

Example

The model snippet

```plaintext
constraint x = 0 \(\lor\) (low \(\le\) x \(\lor\) x \(\le\) up);
```

with parameters \(\text{low}\) and \(\text{up}\), should be reformulated as:

```plaintext
constraint x in \{0\} union low..up;
```

or, even better in this particular case, as:

```plaintext
var \{0\} union low..up: x;
```

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

The model snippet

```
constraint x=3 \/ x=5 \/ x=7;
```

flattens into the very inefficient code:

```
var int: x;
var bool: B3; var bool: B5; var bool: B7;
constraint array_bool_or([B3,B5,B7],true);
constraint B3 -> x=3; % int_eq_imp(x,3,B3)
constraint B5 -> x=5; % this is called
constraint B7 -> x=7; % half-reification
```

It should be reformulated as `constraint x in {3,5,7}`
or, even better, as `var {3,5,7}: x`, which both flatten
into the latter: a domain is a disjunction of candidate values!
Example

The model snippet

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

can be reformulated as (recall that `bool2int(true)=1`):

```plaintext
constraint x = 9 * b;
```
or as (note that array indexing starts by default at 1):

```plaintext
constraint x = [0,9][1+b];
```

But beware of such premature fine-tuning of a model!
The following versions are clearer and often good enough:

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

```plaintext
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 (quantified) formulation.

Example (The all_different constraint predicate)

The constraint `all_different(A)` for an array `A` of size `n` usually leads to faster solving than its definition by a conjunction of \( \frac{n(n-1)}{2} \) disequality constraints:

\[
\text{forall}(i, j \in \text{index_set}(A) \text{ where } i < j) (A[i] \neq A[j])
\]

For more examples, see Topic 3: Constraint Predicates.
Outline

1. The MiniZinc Language
2. Modelling
3. Set Variables & Constraints
4. Modelling Checklist
Motivating Example 1

Example (Agricultural experiment design, AED)

<table>
<thead>
<tr>
<th></th>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>corn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>millet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rye</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spelt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wheat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th></th>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>corn</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>millet</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>oats</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>rye</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>spelt</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>wheat</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

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.
In a BIBD, the plots are **blocks** and the grains are **varieties**:

Example (BIBD *integer* model: √ ⟷ 1 and − ⟷ 0)

```
enum Varieties; enum Blocks;
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([BIBD[v,b]*BIBD[w,b] | b in Blocks]));
```

Example (Instance data for our AED)

```
Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
blockSize = 3; sampleSize = 3; balance = 1;
```
### Example (Idea for another BIBD model)

<table>
<thead>
<tr>
<th>Grain</th>
<th>Plot Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>{plot1, plot2, plot3}</td>
</tr>
<tr>
<td>corn</td>
<td>{plot1, plot4, plot5}</td>
</tr>
<tr>
<td>millet</td>
<td>{plot1, plot6, plot7}</td>
</tr>
<tr>
<td>oats</td>
<td>{plot2, plot4, plot6}</td>
</tr>
<tr>
<td>rye</td>
<td>{plot2, plot5, plot7}</td>
</tr>
<tr>
<td>spelt</td>
<td>{plot3, plot4, plot7}</td>
</tr>
<tr>
<td>wheat</td>
<td>{plot3, plot5, plot6}</td>
</tr>
</tbody>
</table>

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

-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties] of var set of Blocks: BIBD;
0 solve satisfy;
1 constraint forall(b in Blocks)
   (blockSize = sum(v in Varieties)(b in BIBD[v]));
2 constraint forall(v in Varieties)
   (sampleSize = card(BIBD[v]));
3 constraint forall(v, w in Varieties where v < w)
   (balance = card(BIBD[v] intersect BIBD[w]));

Example (Instance data for our AED)

-3 Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
-2 blockSize = 3; sampleSize = 3; balance = 1;
Motivating Example 2²

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 \( d \).

²Based 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:

```plaintext
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 given 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: $S_1 \subseteq S_2 \lor \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$
- Strict order: $S_1 \subset S_2 \lor \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$

where the $S_i$ are set variables and $n$ is an integer variable. Set variables might not pay off in M4CO assignments.
Beware of Variable Integer Ranges

Reification and set variables may appear while flattening complex expressions (which may look non-complex):

Example

```plaintext
var 1..5: x;  array[1..7] of var 1..9: A;
constraint forall(i in 1..x) (A[i]<3);

flattens into inefficient code, linear in the domain size of x:

var 1..5: x;  array[1..7] of var 1..9: A;
var set of 1..5: S;  % prefix of indices i with A[i]<3
var bool: B2; ...; var bool: B9;
constraint 1 in S; constraint A[1] < 3;
constraint B2 <-> 2 in S; constraint B2 <-> 2 <= x;
constraint ...;
constraint B5 <-> 5 in S; constraint B5 <-> 5 <= x;
```

Avoid ranges $\alpha..\beta$ where $\alpha$ or $\beta$ (or both) are variables.
Collection Variables

Definition

- A set variable has a set of sets as domain.
- An array variable has a set of arrays as domain.
- A string variable has a set of strings as domain.

MiniZinc currently has no syntax for the declaration of array variables and string variables, and MiniZinc currently has no constraint predicates for such variables (but our proposal at LOPSTR 2016 and CP-AI-OR 2017 for string variables is ready for integration).

Subtle difference with imperative & OO programming!

array of variables = variable array ≠ array variable

array[int] of var int: A ≠ var array[int] of int: X
A[i] is a variable; A is not a variable; X would be a variable
Outline

1. The MiniZinc Language
2. Modelling
3. Set Variables & Constraints
4. Modelling Checklist
Conventions of all Slides (recommended!)

- Scalar identifiers (enum items, int) start in lowercase
- Mass identifiers (array, enum, set) start in uppercase
- Arrays have self-explanatory total-function identifiers:
  a given|unknown total function $f : X \rightarrow Y$ can be modelled as $\text{array}[X] \text{ of } \text{par}|\text{var } Y : F$
- Index identifiers are lowercase mnemonic: memory aid
- Comments about the next line end in : like line 2 below

Example

1. int: nQueens; % the given number of queens
2. % Row[c] = the row number of the queen in column c:
3. array[1..nQueens] of var 1..nQueens: Row;

Variable $Row[c]$ is like $Row(c)$, denoting the function $Row$ applied to argument $c$. The array $Row$ is not a variable, but an array of variables: it contains row numbers, but calling it $Rows$ would make $Rows[c]$ seem to denote a set of rows!
Ideas for Debugging & Accelerating a Model

- If there are no solutions (or missing solutions) to a known-to-be satisfiable instance, then:
  - Comment away constraints in order to increase the solution set and thereby find unsatisfiable constraints.
  - In the IDE or CLI, choose findMUS as a backend in order to find a minimal unsatisfiable subset (MUS) of constraints: see Section 3.8 of the MiniZinc Handbook.

- In the IDE, choose Run > Profile compilation to see per model line the numbers of constraints & variables generated by its flattening, and the flattening time: if some of these numbers are extreme, then you probably ran afoul of items of the checklist on the next slide.

- In the IDE, choose Run > Compile to see the flat code.
Checklist for Designing or Reading a Model

1. Use `enum` instead of $\alpha\.\beta$ for index ranges when fitting
2. Beware of decision variables without tight domains
3. No explicit variables of type `opt` $\tau$ are used (in M4CO)
4. The index ranges of all arrays start from 1, for clarity
5. No `sum|forall(i in 1..x)` with a var. $x$ is used
6. Beware of `where` $\theta$ and `if` $\theta$ with $\theta$ having variables
7. Beware of explicit (`$\leftarrow$`) & implicit (`(...`) reification
8. Beware of negation and disjunction: `not`, `\\`, `exists`, `xor`, `xorall`, `if \theta then \phi else \psi endif`, `$\leftarrow$, `\rightarrow`, `$\leftarrow$
9. Beware of arbitrarily nested logical quantifications, such as `forall(...exists(...forall(...)...))`
10. Beware of nonlinear, `pow`, `div`, `/`, `mod` constraints