Outline

1. Modelling Viewpoints
2. Implied Constraints
3. Redundant Variables & Channelling Constraints
4. Pre-Computation
Outline

1. Modelling Viewpoints

2. Implied Constraints

3. Redundant Variables & Channelling Constraints

4. Pre-Computation
Example (Objects, Shapes, and Colours)

There are $n$ objects, $s$ shapes, and $c$ colours, with $s \geq n$. Assign a shape and a colour to each object such that:

1. the objects have distinct shapes;
2. the numbers of objects of the used colours are distinct;
3. other constraints, yielding NP-hardness, are satisfied.

This problem can be modelled from different viewpoints:

- Which colour, if any, does each shape have?
- Which shapes, if any, does each colour have?
- Which shape and colour does each object have?
- . . .

Each viewpoint comes with benefits and drawbacks.
Modelling Viewpoints

Implied Constraints

Redundant Variables & Channelling Constraints

Pre-Computation

Example (Objects, Shapes, and Colours)

Viewpoint 1: Which colour, if any, does each shape have?

```plaintext
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours

constraint assert(s >= n, "Not enough shapes");

array[1..s] of var 0..c: Colour; % 0 is a dummy colour

% There are n objects:
constraint count(Colour, 0) = s-n;

% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(global_cardinality(Colour, 1..c));

% The objects have distinct shapes:
% implied by lines 6 and 9
% ... add here the other constraints ...

solve satisfy;
```

Colour 0 is used when there is no object of the given shape. So what are the shape and colour of a particular object?!

Map the objects onto the shapes with a non-0 colour!
Example (Objects, Shapes, and Colours)

Viewpoint 2: Which shapes, if any, does each colour have?

```plaintext
1 int: n; % number of objects
2 int: s; % number of shapes
3 int: c; % number of colours
4 constraint assert(s >= n, "Not enough shapes");
5
6 array[1..c] of var set of 1..s: Shapes;
7
8 % There are n objects:
9 constraint n = sum(colour in 1..c)(card(Shapes[colour]));
10 % The numbers of objects of the used colours are distinct:
11 constraint alldifferent_except_0(colour in 1..c)
     (card(Shapes[colour]));
12 % The objects have distinct shapes:
13 constraint n = card(array_union(Shapes));
14 % ... add here the other constraints ...
15 solve satisfy;
```

Post-process: map the objects onto actually used shapes. Can we also model this viewpoint without set variables? Yes, see the next slide!
Example (Objects, Shapes, and Colours)

Viewpoint 2: Which shapes, if any, does each colour have?

1 int: n; % number of objects
2 int: s; % number of shapes
3 int: c; % number of colours
4 constraint assert(s >= n, "Not enough shapes");
5
6 array[1..c,1..s] of var 0..1: NbrObjects;

7 % There are n objects:
8 constraint n = sum(colour in 1..c, shape in 1..s) (NbrObjects[colour,shape]);
9 % The numbers of objects of the used colours are distinct:
10 constraint alldifferent_except_0(colour in 1..c) (sum(shape in 1..s) (NbrObjects[colour,shape]));
11 % The objects have distinct shapes:
12 constraint forall(shape in 1..s) (1 >= sum(colour in 1..c) (NbrObjects[colour,shape]));
13 % ... add here the other constraints ...
14 solve satisfy;

Which model for viewpoint 2 is clearer or better? Ask & try!
Example (Objects, Shapes, and Colours)

Viewpoint 3: Which shape & colour does each object have?

```plaintext
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");

array[1..n] of var 1..s: Shape;
array[1..n] of var 1..c: Colour;

% There are n objects:
% implied by lines 6 and 7
% The numbers of objects of the used colours are distinct:
constraint
   alldifferent_except_0(global_cardinality(Colour,1..c));
% The objects have distinct shapes:
constraint alldifferent(Shape);
% ... add here the other constraints ...
solve satisfy;
```

We have used two parallel arrays with the same index set but different domains in order to represent pair variables.
Outline

1. Modelling Viewpoints

2. Implied Constraints

3. Redundant Variables & Channelling Constraints

4. Pre-Computation
Example (The Magic Series Problem)

The element at index \( i \) in \( I = 0..(n-1) \) is the number of occurrences of \( i \). Solution: Magic = [1,2,1,0] for \( n=4 \).

Variables: Magic = \[
\begin{array}{cccc}
0 & 1 & \cdots & n-1 \\
\in 0..n & \in 0..n & \cdots & \in 0..n
\end{array}
\]

Constraint:

\[
\text{forall}(i \text{ in } I) (\text{Magic}[i] = \sum (j \text{ in } I) (\text{bool2int} (\text{Magic}[j]=i)))
\]

or, logically equivalently but better:

\[
\text{forall}(i \text{ in } I) (\text{count} (\text{Magic}, i, \text{Magic}[i]))
\]

or, logically equivalently and even better:

\[
\text{global_cardinality_closed} (\text{Magic}, I, \text{Magic})
\]

Implied Constraint:

\[
\sum (\text{Magic}) = n \lor \sum (i \text{ in } I) (\text{Magic}[i] \times i) = n
\]

For \( n=80 \), using a CP solver: only 7 search nodes are explored instead of 302; the solving is 1,000 times faster.
Definition

An implied constraint, also called a redundant constraint, is a constraint that logically follows from other constraints.

**Benefit:**
Solving may be faster, without losing any solutions. However, not all implied constraints accelerate the solving.

**Good practice in MiniZinc:**
Flag implied constraints using the `implied_constraint` predicate. This allows backends to handle them differently, if wanted (see Topic 9: Modelling for CBLS):

```mini
predicate implied_constraint(var bool: c) = c; vs
predicate implied_constraint(var bool: c) = true;
```

**Example**

```mini
constraint implied_constraint(sum(Magic) = n);
```

In Topic 7: Symmetry, we will see the equally recommended `symmetry_breaking_constraint` predicate.
Outline

1. Modelling Viewpoints
2. Implied Constraints
3. Redundant Variables & Channelling Constraints
4. Pre-Computation
Redundant Decision Variables

Example (n-queens)

Use both the $n^2$ decision variables $\text{Queen}[i, j]$ in $0..1$ and the $n$ decision variables $\text{Row}[q]$ in $1..n$.

Definition

A redundant decision variable is a decision variable that represents information that is already represented by some other decision variables. It reflects a different viewpoint.

Benefit: Easier modelling of some constraints, or faster solving, or both.

Examples (see Topic 6: Case Studies)

- Model of Black-Hole Patience
- Models 1 & 3 of Warehouse Location Problem
Channelling Constraints

Example (\(n\)-queens)

Channelling between the \(n\) decision variables \(\text{Row}[i]\) in \(1..n\) and the \(n^2\) decision variables \(\text{Queen}[i,j]\) in \(0..1\):

\[
\forall (i \in 1..n) \left( \text{Row}[i] = \sum (j \in 1..n) (j \times \text{Queen}[i,j]) \right)
\]

Definition

A channelling constraint establishes the coherence of the values of mutually redundant decision variables.

Examples (see Topic 6: Case Studies)

- Model of Black-Hole Patience
- Models 1 & 3 of Warehouse Location Problem
- Experiment with channelling between the viewpoints for the \textit{Objects, Shapes, and Colours} problem (slide 4).
Outline

1. Modelling Viewpoints
2. Implied Constraints
3. Redundant Variables & Channelling Constraints
4. Pre-Computation
Example (Prize-Pool Division)

Consider a maximisation problem where the objective function is the division of an unknown prize pool by an unknown number of winners:

```plaintext
... 
array[1..5] of int: Pools = [1000,5000,15000,20000,25000];
var 1..5: x;
var 1..500: numWinners;
...
solve maximize Pools[x] div numWinners; % implicit: element!
```

**Observation:** We should avoid using the `div` function on decision variables, because:

- It yields weak **inference**, at least in CP & LCG solvers.
- Its **inference** takes unnecessary time and memory.
- It is not supported by all MiniZinc backends.

**Idea:** We can pre-compute all possible objective values.
Idea: We can pre-compute all possible objective values.

Example (Prize-Pool Division, revisited)

Pre-compute a 2D array, indexed by 1..5 and 1..500, for each possible value pair of x and numWinners:

```plaintext
... 
array[1..5] of int: Pools = [1000,5000,15000,20000,25000];  
var 1..5: x;  
var 1..500: numWinners;  
...  
array[1..5,1..500] of int: objFun = array2d(1..5,1..500,  
    [Pools[p] div n | p in 1..5, n in 1..500]);  
solve maximize objFun[x,numWinners]; % implicit: 2D-element!
```