DM841

Constraint Programming

Lecture 6

Global Constraints

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Christian Schulte, KTH Royal Institute of Technology*]

Resume

- Examples
  - graph labelling with consecutive numbers
  - Cryptarithmetic (or verbal arithmetic or cryptarithm): Send + More = Money
  - Graph (map) coloring
  - Production planning
  - Investment planning
  - Job shop scheduling
  - Stable marriage problem
  - Social golfers
  - Grocery
  - Magic series
- Language elements:
  - parameters and variables, data types and enumerations
  - relational operators
  - arithmetics operators and functions (integer and float)
  - basic structure
  - arrays, sets, comprehensions
  - aggregate functions
  - set variables

Outline

Predicates and Functions in MiniZinc

- A predicate is a function with output type `var bool`
- Predicates capture complex constraints in a succinct way.
- They can be built-in or user defined
- They let the modeller to define his/her own high-level constraints for *re-use* and *modularization*
- ultimately they allow the development of application specific libraries defining the standard constraints and types.
- Global constraints in MiniZinc: `https://www.minizinc.org/doc-latest/en/lib-globals.html` and the solver specific `globals.mzn`

Defining Predicates

```
predicate <pred-name> ( <arg-def>, ..., <arg-def> ) = <bool-exp>
```

`<arg-def>` type declaration (in arg-def the index types for arrays can be unbounded, eg `array[int]`)

```
test <pred-name> ( <arg-def>, ..., <arg-def> ) = <bool-exp>
```

new constraints that only involve parameters. Useful to write fixed tests for a conditional expression. Example:

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

```
assert ( <bool-exp>, <string-exp>, <exp> )
```

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.

```
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] );
```

```
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 JOB) (
     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 TASK) (
        forall(i,k in JOB where i < k) (
            no_overlap(s[i,j], d[i,j], s[k,j], d[k,j])
            )
    );
```

Defining Functions

Similar to predicates, but with a more general return type.
Useful for defining and documenting complex expressions that are used multiple times in a model.

```
function <ret-type> : <func-name> ( <arg-def>, ..., <arg-def> ) = <exp>
```

```
function var int: manhattan(var int: x1, var int: y1,
                              var int: x2, var int: y2) =
            abs(x1 - x2) + abs(y1 - y2);
```

Reflection Functions

for generic tests and predicates

return information about array index sets, var set domains and decision variable ranges.

```
index_set(<1-D array>)
index_set_1of2(<2-D array>)
index_set_2of2(<2-D array>)
...
```

Scopes

MiniZinc has a single namespace, so all variables appearing in declarations are visible in every expression in the model.
Locally scoped variables can be achieved:

- as iterator variables in comprehension expressions
- as predicate and function arguments
- using let expressions

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

`let`

Let expressions are a facility to introduce new variables. This is useful for creating common sub expressions, and for defining local variables for predicates.

In FlatZinc they are transformed in the creation of new variables.

Option Type

An option type variable can take the additional value <> indicating absent.
An option type variable is declared as:

```
var opt <type> : <var-name>
```

where `<type>` is one of int, float or bool or a fixed range expression.
Option type variables can be parameters.

Most of the previously seen functions and operators work with option type variables. (See manual also for weaker typ, eg, ~=)

```
array[1..n] of var int: x;
constraint forall(i in 1..n where x[i] >= 0)(x[i] <= limit);
```

Tuple and Records

Handling “objects”:
It is common practice in MiniZinc to have different arrays for different types of data points or decisions. These arrays will then share a common index mapping to which object each data point belongs.

Alternatively:

```
<var-par> tuple(<ti-expr>, ...): <var-name>
```

```
<var-par> record(<ti-expr-and-id>, ...): <var-name>
```

```
tuple(int, bool): t;
t = (1, true);
var tuple(int, bool): y; % same as tuple(var int,
     var bool))
any: x = (1, true, 2.0);
x.1 % 1
x.3 % 2.0
```

```
record(int: i, bool: b): r;
r = (i: 1, b: true);
var record(int: i, bool: b): y % same as record(var
     int: i, var bool: b)
any: x = (i: 1, b: true)
x.i % 1
x.b % true
```

Synomyms

```
type <ident> <annotations> = <ti-expr>;
```

```
type Coord = var record(int: x, int: y, int: z);
type Number = int;
```

```
array[1..10] of Coord: placement;
function Number:add(Number: x, Number: y) = x + y;
```

Mixing Variables and Parameters

The left approach has troubles importing data from files. Prefer the approach on the right.

```
enum EmpId;
type Employee = record(
  string: name,
  array[Timespan] of bool: available,
  set of Capability: capacities,
  array[Timespan] of var Shift: shifts,
  var 0..infinity: hours,
);
array[EmpId] of Employee: employee;
```

```
enum EmpId;
type EmployeeData = record(
  string: name,
  array[Timespan] of bool: available,
  set of Capability: capacities,
);
type EmployeeVar = record(
  array[Timespan] of var Shift: shifts,
  var 0..infinity: hours,
);
array[EmpId] of EmployeeData: employee_data;
array[EmpId] of EmployeeVar: employee_var;

type Employee = EmployeeData ++ EmployeeVar;
array[EmpId] of Employee: employee = [employee_data[
     id] ++ employee_var[id] | id in EmpId];
```

Packing rectangles

```
type Dimensions = record(int: width, int: height);
type Coordinates = record(var 0..infinity: x, var 0..infinity: y);
```

```
type Rectangle = Dimensions ++ Coordinates;
% No overlap predicate
predicate no_overlap(Rectangle: rectA, Rectangle: rectB) =
  rectA.x + rectA.width <= rectB.x
  \/ rectB.x + rectB.width <= rectA.x
  \/ rectA.y + rectA.height <= rectB.y
  \/ rectB.y + rectB.height <= rectA.y;
```

```
% Instance data
array[_] of Dimensions: rectDim;
Dimensions: area;

% Decision variables
array[index_set(rectDim)] of Coordinates: rectCoord ::no_output;

array[_] of Rectangle: rectangles ::output = [ rectDim[i] ++ rectCoord[i] | i in index_set(rectDim)];

% Constraint: rectangles must be placed within the area
constraint forall(rect in rectangles) (
     rect.x + rect.width <= area.width
     /\ rect.y + rect.height <= area.height
);
% Constraint: no rectangles can overlap
constraint forall(i, j in index_set(rectangles) where i < j) (
     no_overlap(rectangles[i], rectangles[j])
```

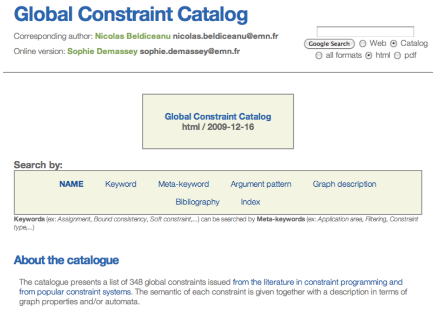
Packing rectangles

```
{
  "area": {
    "width": 5,
    "height": 5
  },
  "rectDim": [
    {
      "width": 3,
      "height": 3
    },
    {
      "width": 2,
      "height": 1
    },
    {
      "width": 1,
      "height": 2
    },
    {
      "width": 2,
      "height": 2
    },
    {
      "width": 2,
      "height": 3
    }
  ]
}
```

Outline

Global Constraints Catalog

- https://sofdem.github.io/gccat/ by Nicolas Beldiceanu, Mats Carlsson and Sophie Demassey
- Earlier version https://web.imt-atlantique.fr/x-info/sdemasse/gccatold/



Global Constraint Catalog

Corresponding author: Nicolas Beldiceanu nicolas.beldiceanu@emn.fr

Online version: Sophie Demassey sophie.demassey@emn.fr

Global Constraint Catalog
html / 2009-12-16

Search by:

Keywords (ex: *Assignment, Bound consistency, Soft constraint,...*) can be searched by **Meta-keywords** (ex: *Application area, Filtering, Constraint type,...*)

About the catalogue

The catalogue presents a list of 348 global constraints issued from the literature in constraint programming and from popular constraint systems. The semantic of each constraint is given together with a description in terms of graph properties and/or automata.

Constraint Satisfaction Model

Constraint Satisfaction Problem (CSP)

A CSP is

- a finite set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ with domain extension $\mathcal{D} = D(x_1) \times \cdots \times D(x_n)$, together with
- a finite set of constraints $\mathcal{C} = \{C_1, \ldots, C_m\}$, each on a subset of $\mathcal{X}$.

A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in \mathcal{X}$, such that all constraints are satisfied simultaneously.

Global Constraint: Alldifferent

Global constraint:

set of more elementary constraints that exhibit a special structure when considered together.

`alldifferent` constraint

Let $x_1, x_2, \ldots, x_n$ be variables. Then:

$$\texttt{alldifferent}(x_1, ..., x_n) = \{(d_1, ..., d_n) \mid \forall i \; d_i \in D(x_i), \quad \forall i \neq j, \; d_i \neq d_j\}.$$

Constraint arity: number of variables involved in the constraint

Note: different notation and names used in the literature
In Gecode `distinct`

Alldifferent in MiniZinc

The `alldifferent` constraint takes an array of integer variables and constrains them to take different values. Form

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

Global Constraint: Sum

Sum constraint

Let $x_1, x_2, \ldots, x_n$ be variables. To each variable x_i, we associate a scalar $c_i \in \mathbb{Q}$. Furthermore, let z be a variable with domain $D(z) \subseteq \mathbb{Q}$. The `sum` constraint is defined as

$$\texttt{sum}([x_1, \ldots, x_n], z, c) = \left\{ (d_1, \ldots, d_n, d) \mid \forall i, d_i \in D(x_i), d \in D(z), d = \sum_{i=1,\ldots,n} c_i d_i \right\}.$$

In Gecode: `linear(home, x, IRT_GR, c)`

```
linear(Home home, const IntArgs &a, const IntVarArgs &x,
IntRelType irt, IntVar y, IntConLevel icl=ICL_DEF)
```

Global Constraint: Sum

```
predicate sum_pred(var int: i, array [int] of set of int: sets, array [int] of int: cs, var int: s)
```

Requires that the sum of `cs[i1]..cs[iN]` equals `s`, where `i1..iN` are the elements of the `i`th set in `sets`.
Also possible:
`s = sum(i in index_set(x)) (coeffs[i]*x[i])`
Nb: not called **`sum`** as in the constraints catalog because **`sum`** is a MiniZinc built-in function.

Global Constraint: Knapsack

Knapsack constraint

Rather than constraining the sum to be a specific value, the knapsack constraint states the sum to be within a lower bound l and an upper bound u, i.e., such that $D(z) = [l, u]$. The knapsack constraint is defined as

$$\text{knapsack}([x_1, \ldots, x_n], z, c) =$$

$$\left\{ (d_1, \ldots, d_n, d) \mid d_i \in D(x_i)\, \forall i, d \in D(z), d \leq \sum_{i=1,\ldots,n} c_i d_i \right\} \cap$$

$$\left\{ (d_1, \ldots, d_n, d) \mid d_i \in D(x_i)\, \forall i, d \in D(z), d \geq \sum_{i=1,\ldots,n} c_i d_i \right\}.$$

$$\min D(z) \leq \sum_{i=1,\ldots,n} c_i x_i \leq \max D(z)$$

In Gecode: `linear(Home home, const IntArgs &a, const IntVarArgs &x, IntRelType irt, IntVar y, IntConLevel icl=ICL_DEF)`
In Minizinc:

```
predicate knapsack(array [int] of int: w,
                   array [int] of int: p,
                   array [int] of var int: x,
                   var int: W,
                   var int: P)
```

or
`s = sum(i in index_set(x)) (coeffs[i]*x[i])`

Global Constraint: `cardinality`

`cardinality` or `gcc` (global cardinality constraint)

Let $x_1, \dots, x_n$ be assignment variables whose domains are contained in $\{v_1, \dots, v_{n'}\}$ and let $\{c_{v_1}, \dots, c_{v_{n'}}\}$ be count variables whose domains are sets of integers. Then

$$\texttt{cardinality}([x_1, \ldots, x_n], [c_{v_1}, \ldots, c_{v_{n'}}]) = \{(w_1, \ldots, w_n, o_1, \ldots, o_{n'}) \mid w_j \in D(x_j)\, \forall j, \; \texttt{occ}(v_i, (w_1, \ldots, w_n)) = o_i \in D(c_{v_i})\, \forall i\}.$$

(`occ` number of occurrences)

$\rightsquigarrow$ generalization of `alldifferent`

In Gecode: `count`

Counting constraints in MiniZinc

restrict how many times certain values occur in an array of variables.

```
count(i in x)(i=c) <= d
```

However, if your model contains multiple counting constraints over the same array:

```
predicate distribute(array [int] of var int: card, array [int] of var int: value, array [int] of var int:
     base)
```

Requires that `card[i]` is the number of occurrences of `value[i]` in `base`. The values in value need not be distinct.

```
predicate global_cardinality(array [int] of var int: x, array [int] of int: cover, array [int] of var int:
     counts)
```

Requires that the number of occurrences of parameter `cover[i]` in `x` is `counts[i]`.

Global Constraint: among and sequence

among

Let $x_1, \ldots, x_n$ be a tuple of variables, S a set variable, and l and u two nonnegative integers

$$\texttt{among}([x_1, ..., x_n], S, l, u)$$

At least l and at most u of variables take values in S.
In Gecode: `count`

sequence

Let $x_1, \ldots, x_n$ be a tuple of variables, S a set variable, and l and u two nonnegative integers, s a positive integer.

$$\texttt{sequence}([x_1, ..., x_n], S, l, u, s)$$

At least l and at most u of variables take values from S in s consecutive variables

Car Sequencing Problem

Car Sequencing Problem

- an assembly line makes 50 cars a day
- 4 types of cars
- each car type is defined by options: {air conditioning, sun roof}

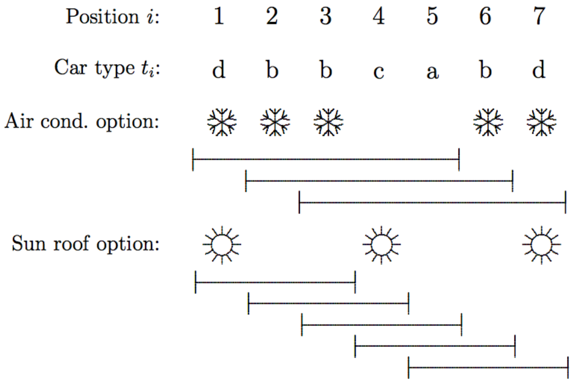
type	air cond.	sun roof	demand
a	no	no	20
b	yes	no	15
c	no	yes	8
d	yes	yes	7

- at most 3 cars in any sequence of 5 can be given air conditioning
- at most 1 in any sequence of 3 can be given a sun roof

Task: sequence the car types so as to meet demands while observing capacity constraints of the assembly line.

Car Sequencing Problem

Sequence constraints



Car Sequencing Problem: CP model

Car Sequencing Problem

Let t_i be the decision variable that indicates the type of car to assign to each position i in the sequence.

$$\begin{aligned}
&\text{cardinality}([t_1, \ldots, t_{50}], (a, b, c, d), (20, 15, 8, 7), (20, 15, 8, 7))\\
&\text{among}([t_i, \ldots, t_{i+4}], \{b, d\}, 0, 3), \qquad \forall i = 1..46\\
&\text{among}([t_i, \ldots, t_{i+2}], \{c, d\}, 0, 1), \qquad \forall i = 1..48\\
&t_i \in \{a, b, c, d\}, i = 1, \ldots, 50.
\end{aligned}$$

Note: in Gecode `among` is `count`.
However, we can use `sequence` for the two `among` constraints above:

$$\begin{aligned}
&\text{sequence}([t_1, \ldots, t_{50}], \{b, d\}, 0, 3, 5),\\
&\text{sequence}([t_1, \ldots, t_{50}], \{c, d\}, 0, 1, 3),
\end{aligned}$$

Global Constraint: `nvalues`

`nvalues`

Let $x_1, \ldots, x_n$ be a tuple of variables, and l and u two nonnegative integers

$$\texttt{nvalues}([x_1, ..., x_n], l, u)$$

At least l and at most u different values among the variables

$\rightsquigarrow$ generalization of `alldifferent`
In Gecode: `nvalues`
In MiniZinc `predicate nvalue(var int: n, array [$X] of var int: x)`

Global Constraint: `stretch`

`stretch` (In Gecode: via `regular` and `extensional`)

Let $x_1, \dots, x_n$ be a tuple of variables with finite domains,
v an m-tuple of possible values of the variables,
l an m-tuple of lower bounds and u an m-tuple of upper bounds.
A stretch is a maximal sequence of consecutive variables that take the same value, i.e., $x_j, \dots, x_k$ for v if $x_j = \dots = x_k = v$ and $x_{j-1} \neq v$ (or $j = 1$) and $x_{k+1} \neq v$ (or $k = n$).

`stretch`$([x_1, ..., x_n], \mathrm{v}, \mathrm{l}, \mathrm{u})$ `stretch-cycle`$([x_1, ..., x_n], \mathrm{v}, \mathrm{l}, \mathrm{u})$

for each $j \in \{1, \dots, m\}$ any stretch of value v_j in x have length at least l_j and at most u_j.

In addition:

`stretch`$([x_1, ..., x_n], \mathrm{v}, \mathrm{l}, \mathrm{u}, P)$

with P set of patterns, i.e., pairs $(v_j, v_{j'})$. It imposes that a stretch of values v_j must be followed by a stretch of value $v_{j'}$

Global Constraint: `element`

“`element`” constraint

Let y be an integer variable,
z a variable with finite domain,
and c an array of constants, i.e., $c = [c_1, c_2, \ldots, c_n]$.
The element constraint states that z is equal to the y-th variable in c, or $z = c_y$.
More formally:

$$\texttt{element}(y, z, [c_1, \ldots, c_n]) = \{(e, f) \mid e \in D(y), f \in D(z), f = c_e\}.$$

In MiniZinc

```
array[1..6] of int: c = {5,1,4,9,16,25};
var int: y;
var int: z;
z=c[y];
```

Assignment problems

The assignment problem is to find a minimum cost assignment of m tasks to n workers ($m \leq n$). Each task is assigned to a different worker, and no two workers are assigned the same task.
If assigning worker i to task j incurs cost c_{ij}, the problem is simply stated:

$$\begin{aligned} \min \quad & \sum_{i=1,\ldots,n} c_{ix_i} \\ & \texttt{alldiff}([x_1, \ldots, x_n]), \\ & x_i \in D_i, \forall i = 1, \ldots, n. \end{aligned}$$

Note: cost depends on position. Recall: with $n = m$ min weighted bipartite matching (Hungarian method)
with supplies/demands transshipment problem

Global Constraint: `channel`

"`channel`" constraint

Let y be array of integer variables, and x be an array of integer variables:

$$\texttt{channel}([y_1, \ldots, y_n], [x_1, \ldots, x_n]) = \\ \{([e_1, \ldots, e_n], [d_1, \ldots, d_n]) \mid e_i \in D(y_i), \forall i, d_j \in D(x_j), \forall j, e_i = j \wedge d_j = i\}.$$

```
predicate inverse(array [int] of var int: f, array [int] of var int: invf)
```

Constrains two arrays of int variables, `f` and `invf`, to represent inverse functions. All the values in each array must be within the index set of the other array

Employee Scheduling problem

Four nurses are to be assigned to eight-hour shifts.
Shift 1 is the daytime shift, while shifts 2 and 3 occur at night.
The schedule repeats itself every week. In addition,

1. Every shift is assigned exactly one nurse.
2. Each nurse works at most one shift a day.
3. Each nurse works at least five days a week.
4. To ensure a certain amount of continuity, no shift can be staffed by more than two different nurses in a week.
5. To avoid excessive disruption of sleep patterns, a nurse cannot work different shifts on two consecutive days.
6. Also, a nurse who works shift 2 or 3 must do so at least two days in a row.

Employee Scheduling problem

Feasible Solutions

Solution viewed as assigning workers to shifts.

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Shift1	A	B	A	A	A	A	A
Shift2	C	C	C	B	B	B	B
Shift3	D	D	D	D	C	C	D

Solution viewed as assigning shifts to workers.

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Worker A	1	0	1	1	1	1	1
Worker B	0	1	0	2	2	2	2
Worker C	2	2	2	0	3	3	0
Worker D	3	3	3	3	0	0	3

Employee Scheduling problem

Feasible Solutions

Let w_{sd} be the nurse assigned to shift s on day d, where the domain of w_{sd} is the set of nurses $\{A, B, C, D\}$.
Let t_{id} be the shift assigned to nurse i on day d, and where shift 0 denotes a day off.

1. `alldiff`$(w_{1d}, w_{2d}, w_{3d}), d = 1, \ldots, 7$
2. `cardinality`$(W, (A, B, C, D), (5, 5, 5, 5), (6, 6, 6, 6))$
3. `nvalues`$(\{w_{s1}, \ldots, w_{s7}\}, 1, 2), s = 1, 2, 3$
4. `alldiff`$(t_{Ad}, t_{Bd}, t_{Cd}, t_{Dd}), d = 1, ..., 7$
5. `cardinality`$(\{t_{i1}, \ldots, t_{i7}\}, 0, 1, 2), i = A, B, C, D$
6. `stretch-cycle`$((t_{i1}, \ldots, t_{i7}), (2, 3), (2, 2), (6, 6), P), i = A, B, C, D$
7. $w_{t_{id}d} = i, \forall i, d, \quad t_{w_{sd}s} = s, \forall s, d$

Circuit problems

Given a directed weighted graph $G = (N, A)$, find a circuit of min cost:

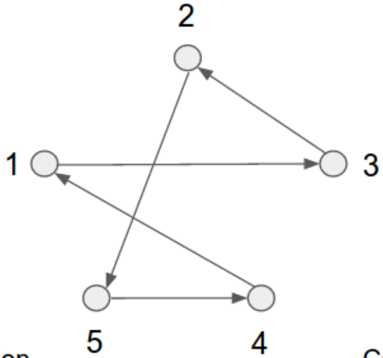
$$\min \sum_{i=1,\dots,n} c_{x_i x_{i+1}}$$
$$\texttt{alldiff}([x_1, \dots, x_n]),$$
$$x_i \in D_i, \forall i = 1, \dots, n.$$

Note: cost depends on sequence.

An alternative formulation is

$$\min \sum_{i=1,\dots,n} c_{i y_i}$$
$$\texttt{circuit}([y_1, \dots, y_n]),$$
$$y_i \in D_i = \{j \mid (i, j) \in A\}, \forall i = 1, \dots, n.$$

Circuit representation



Classical permutation notation

	1	2	3	4	5
x:	1	3	2	5	4

Cauchy two-line notation

	1	2	3	4	5
y:	3	5	2	1	4

Global Constraint: `circuit`

"`circuit`" constraint

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables with respective domains $D(x_i) \subseteq \{1, 2, ..., n\}$ for $i = 1, 2, ..., n$. Then

$$\texttt{circuit}(x_1, \ldots, x_n) = \{(d_1, ..., d_n) \mid \forall i, d_i \in D(x_i), d_1, ..., d_n \text{ is cyclic } \}.$$

Circuit problems - Linking viewpoints

A model with redundant constraints is as follows:

$$
\begin{aligned}
\min \quad & z && (1)\\
& z \geq \sum_{i=1,\dots,n} c_{x_i x_{i+1}} && (2)\\
& z \geq \sum_{i=1,\dots,n} c_{i y_i} && (3)\\
& \texttt{alldiff}([x_1,\dots,x_n]), && (4)\\
& \texttt{circuit}([y_1,\dots,y_n]), && (5)\\
& x_1 = y_{x_n} = 1, \quad x_{i+1} = y_{x_i}, i = 1,\dots,n-1 && (6)\\
& x_i \in \{1,\dots,n\}, \forall i = 1,\dots,n, && (7)\\
& y_i \in D_i = \{j \mid (i,j) \in A\}, \forall i = 1,\dots,n. && (8)
\end{aligned}
$$

Line (6) implements the linking between the two formulations.
In Gecode it can be implemented with the `element`:

```
element(y, x[i], x[i+1])
```

CP Modeling Guidelines [Hooker, 2011]

1. A specially-structured subset of constraints should be replaced by a single global constraint that **captures the structure**, when a suitable one exists. This produces a more succinct model and can allow more effective filtering and propagation.

2. A global constraint should be replaced by a more specific one when possible, to exploit more effectively the special structure of the constraints.

3. The addition of redundant constraints (i..e, constraints that are implied by the other constraints) can improve propagation.

4. When two alternate formulations of a problem are available, including both (or parts of both) in the model may improve propagation.
   Different variables are linked through the use of channeling constraints.

Extensional Constraints: Table

enforces that a tuple (array) of variables takes a value from a set of tuples

```
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)
```

enforces $x \in t$ where we consider x and each row in t to be a tuple, and t to be a set of tuples.

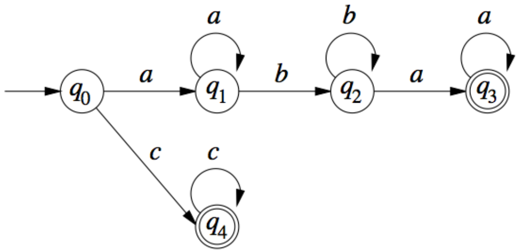
```
array[FOOD,FEATURE] of int: dd; % food database
array[FEATURE] of var int: main;
array[FEATURE] of var int: side;
array[FEATURE] of var int: dessert;

enum FOOD;
FOOD = { icecream, banana, chocolatecake, lasagna, steak, rice, chips, brocolli, beans} ;
enum FEATURE = { name, energy, protein, salt, fat, cost};

dd = [| icecream,      1200,  50,  10, 120,  400
      | banana,         800, 120,   5,  20,  120
      | chocolatecake, 2500, 400,  20, 100,  600
      | lasagna,       3000, 200, 100, 250,  450
      | steak,         1800, 800,  50, 100, 1200
      | rice,          1200,  50,   5,  20,  100
      | chips,         2000,  50, 200, 200,  250
      | brocolli,       700, 100,  10,  10,  125
      | beans,         1900, 250,  60,  90,  150 |];


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

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];
```

Extensional Constraints: Regular

"regular" constraint

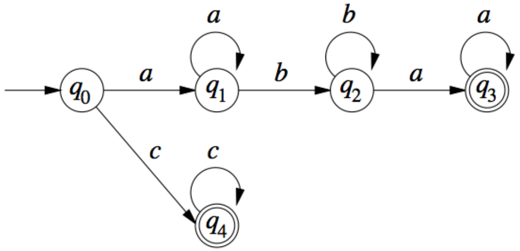
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a Deterministic Finite Automaton (DFA) and let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables with $D(x_i) \subseteq \Sigma$ for $1 \leq i \leq n$. Then

$$\texttt{regular}(X, M) = \{(d_1, ..., d_n) \mid \forall i, d_i \in D(x_i), [d_1, d_2, \ldots, d_n] \in L(M)\}.$$



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a"] \arrow[dr, "c"] & q_1 \arrow[r, "b"] \arrow[loop above, "a"] & q_2 \arrow[r, "a"] \arrow[loop above, "b"] & q_3 \arrow[loop above, "a"] \\ & & q_4 \arrow[loop above, "c"] & & \end{tikzcd}$$

Global Constraint: `regular`



Example

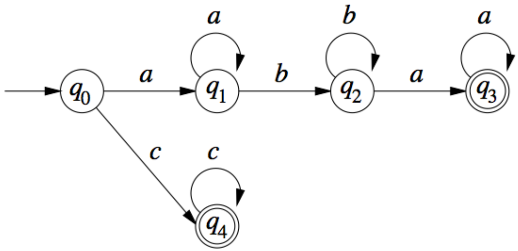
Given the problem

$$x_1 \in \{a, b, c\}, \quad x_2 \in \{a, b, c\}, \quad x_3 \in \{a, b, c\}, \quad x_4 \in \{a, b, c\},$$

$$\texttt{regular}([x_1, x_2, x_3, x_4], M).$$

One solution to this CSP is $x_1 = a, x_2 = b, x_3 = a, x_4 = a$.

Global Constraint: `regular`

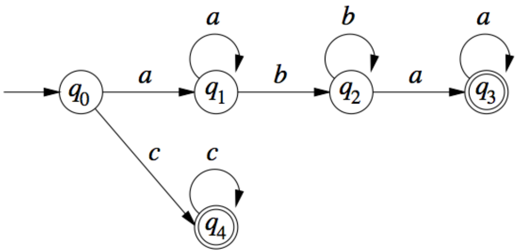


```
regular(array[int] of var int: x, int: Q, int: S, array[int,int] of int: d, int: q0, set of int: F)
```

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`)
- final states `F` (which all must be in `1..Q`). State `0` is reserved to be an always failing state.

Global Constraint: `regular`



	a	b	c
1	2	0	5
2	2	3	0
3	4	3	0
4	4	0	0
5	0	0	5

```
include "globals.mzn";

enum LETTERS = {a,b,c};
array[1..5] of var LETTERS: x;
int: Q = 5;
int: S = card(LETTERS);
int: q0 = 1;
set of int: STATE = 1..5;
set of int: final = {4,5};
```

```
array[STATE,LETTERS] of int: t =
[| 2, 0, 5    % state 1
 | 2, 3, 0    % state 2
 | 4, 3, 0    % state 3
 | 4, 0, 0    % state 4
 | 0, 0, 5|]; % state 5

constraint regular(x, Q, S, t, q0, final);

solve satisfy;
```

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

constraints that:

- the array `x` (which must all be in the range `1..S`)
- is accepted by the NFA of `Q` states with input `1..S` and
- transition function `d` (which maps `<1..Q, 1..S>` to subsets of `1..Q`) and
- initial state `q0` (which must be in `1..Q`) and
- accepting states `F` (which all must be in `1..Q`).

There is no need for a failing state 0, since the transition function can map to an empty set of states.

DFA

```
array[STATE,LETTERS] of int: t =
[| 2, 0, 5    % state 1
 | 2, 3, 0    % state 2
 | 4, 3, 0    % state 3
 | 4, 0, 0    % state 4
 | 0, 0, 5|]; % state 5
```

NFA

```
array[STATE,LETTERS] of set of int: t =
[| {2,3}, {}, {5}    % state 1
 | {2,3}, {3}, {}    % state 2
 | {4}, {3}, {}    % state 3
 | {4}, {}, {}    % state 4
 | {}, {}, {5}|]; % state 5
```

Scheduling Constraints

One job at a time on a machine (disjunctive machines):

"`disjunctive`" scheduling

Let $(x_1, \ldots, x_n)$ be a tuple of (integer/real)-valued variables indicating the starting time of a job j.
Let $(p_1, \ldots, p_n)$ be the processing times of each job.

$$\texttt{disjunctive}([x_1, \ldots, x_n], [p_1, \ldots, p_n]) = \{[s_1, \ldots, s_n] \mid \forall i, j, i \neq j,\ (s_i + p_i \leq s_j) \vee (s_j + p_j \leq s_i)\}$$

In MiniZinc:

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

In Gecode:

```
IntArgs p(4, 2,7,4,11);
unary(home, s, p);
```

Scheduling Constraints

In Resource Constrained Project Scheduling each resource can be used at most up to its capacity:

`cumulative` constraints [Aggoun and Beldiceanu, 1993]

- r_j release time of job j
- p_j processing time
- d_j deadline
- c_j resource consumption
- C limit not to be exceeded at any point in time

Let x be an n-tuple of (integer/real) value variables denoting the starting time of each job

$$\texttt{cumulative}([x_j],[p_j],[c_j],C) := \{([s_j],[p_j],[c_j],C) \mid \forall t \sum_{i \mid s_i \leq t \leq s_i+p_i} c_i \leq C\}$$

With $c_j = 1$ forall j and $C = 1 \rightsquigarrow$ `disjunctive`

Scheduling Constraints: Cumulative

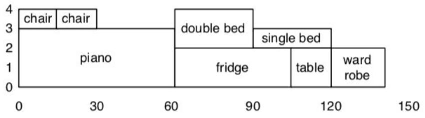
The cumulative constraint is used in scheduling problems for describing cumulative resource usage.

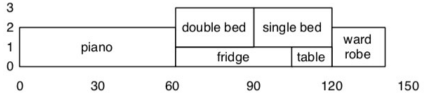
```
cumulative(array[int] of var int: s,
           array[int] of var int: d,
           array[int] of var int: r,
           var int: b)
```

A set of tasks with start times s, durations d, and resource requirements r,
must never require more than a global resource bound b at any one time.

Start times can be optional variables, so that absent tasks do not need to be scheduled.

```
include "cumulative.mzn";

enum OBJECTS;
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 = \(start)\nend = \(end)\n"];
```





Scheduling Constraints

`cumulatives` generalizes `cumulative` by: [Beldiceanu and Carlsson, 2002]

1. allowing to have several cumulative resurces and that each task has to be assigned to one of them
2. the resource consumption by any task is a variable that can take positive or negative values
3. it is possible to enforce the cumulated consumption to be less than or equal, or greater or equal to a given level.
4. the previous point on the cumulated resource consumption is enforced only for those time-points that are overalpped by at least 1 task.
   permitting multiple cumulative resources as well as negative resource consumptions by the tasks.

Scheduling Constraints

Cumulatives

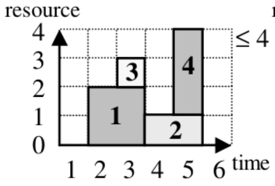
`cumulatives` constraints [Beldiceanu and Carlsson, 2002]

- variables $(y_j, x_j, d_j, c_j, e_j)$ for job $j \in J$
  $y_j \in \mathbb{Z}$ machine; $d_j \in \mathbb{Z}^+$ duration; $x_j \in \mathbb{Z}$ start time; $c_j \in \mathbb{Z}$ consumption; $e_j \in \mathbb{Z}$ end time
- parameters (r, L_r) for resource $r \in R$, L_r limit.
- constraint $\leq$ or $\geq$

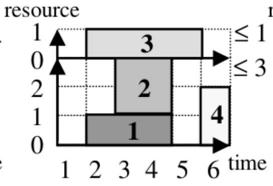
$$\texttt{cumulatives}([y_j], [x_j], [d_j], [c_j], [e_j], [L_r], \lesseqgtr) :=$$

$$\Big\{([q_j], [s_j], [p_j], [u_j], [f_j], [L_{q_j}], \lesseqgtr) \mid$$
$$\forall j \in J: \ s_j + p_j = f_j \quad \text{and}$$
$$\forall j \in J, \forall t \in [s_j, e_j - 1], \hat{r} = y_j :$$
$$\sum_{i \mid \substack{s_i \leq t \leq s_i + p_i \\ y_i = y_j}} c_i \leq L_{\hat{r}} \Big\}$$

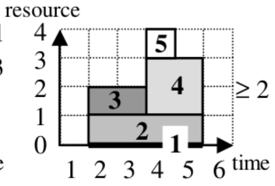
examples of cases modelled by `cumulatives`



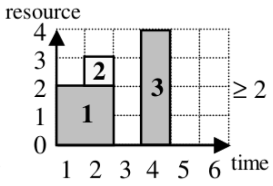
(A)



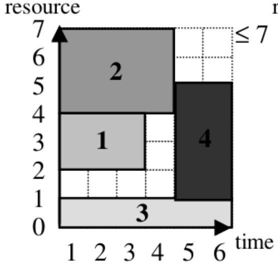
(B)



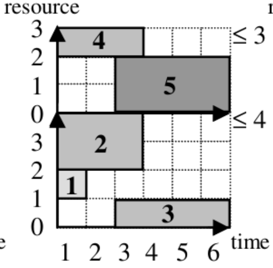
(C)



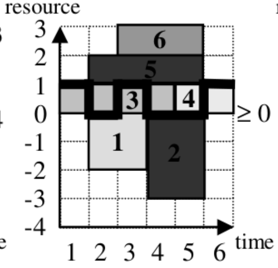
(D)



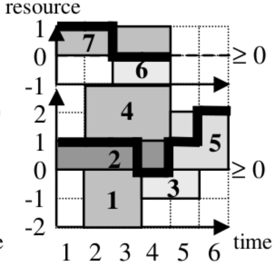
(E)



(F)



(G)



(H)

Others

- Sorted constraints (`sorted`(x, y))
- Bin-packing constraints (`binpacking`(l, b, s))
  l_j is the load variable of bin j, b_i the bin variable of item i, s_i size of item i
- Geometrical packing constraints (`nooverlap`) `diffn`$((x^1, \Delta x^1), \dots, (x^m, \Delta x^m))$ arranges a given set of multidimensional boxes in n-space such that they do not overlap
  (aka, `nooverlap`)
- Value precedence constraints (`precede`(x, s, t))
- Logical implication: `conditional`$(\mathcal{D}, \mathcal{C})$ between sets of constrains $\mathcal{D} \Rightarrow \mathcal{C}$ (`ite`)

More

- `clique`$(x|G,k)$ requires that a given graph contain a clique of size k
- `cycle`$(x|y)$ select edges such that they form exactly y directed cycles in a graph.
- `cutset`$(x|G,k)$ requires that for the set of selected vertices V', the set $V \setminus V'$ induces a subgraph of G that contains no cycles.

References

Beldiceanu N. and Carlsson M. (2002). **A New Multi-resource cumulatives Constraint with Negative Heights**, pp. 63–79. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hooker J.N. (2011). **Hybrid modeling**. In *Hybrid Optimization*, edited by P.M. Pardalos, P. van Hentenryck, and M. Milano, vol. 45 of **Optimization and Its Applications**, pp. 11–62. Springer New York.

van Hoeve W. and Katriel I. (2006). **Global constraints**. In *Handbook of Constraint Programming*, chap. 6. Elsevier.