Topic 3: Constraint Predicates
(Version of 14th October 2019)

Pierre Flener, Gustav Björdal, 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

1Many thanks to Guido Tack for feedback
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lelesseq
11. regular, table
12. Modelling Checklist
Examples

Let $A$ be a 1d array of variables:

- An $\text{alldifferent}(A)$ constraint holds if and only if all the elements of $A$ take different values:

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

- A $\text{count}(A,v) \geq c$ constraint holds if and only if the count of occurrences in $A$ of $v$ is at least $c$, where $v$ and $c$ can be variables:

  \[
  \sum(i \in \text{index_set}(A))(A[i]=v) \geq c
  \]
Definition

A definition of a constraint predicate is its semantics, stated in MiniZinc in terms of usually simpler constraint predicates.

Definition

Each use of a predicate is decomposed during flattening by inlining either its MiniZinc-provided default definition or an overriding backend-provided solver-specific definition.
Motivation:

+ More compact and intuitive models, because more expressive predicates are available: islands of common combinatorial structure are identified in declarative high-level abstractions.

+ Faster solving, due to better *inference* and *relaxation*, enabled by more global information in the model, provided the predicate is a built-in of the used solver.

Enabling constraint-based modelling:

- Constraint predicates over *any* number of variables go by many names: *global-constraint predicates*, *combinatorial-constraint predicates*, ...

Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
**The `alldifferent` Predicate**

### Definition (Laurière, 1978)

An `alldifferent(A)` constraint holds if and only if all the elements of the 1d array `A` of variables take different values.

Its default definition in MiniZinc is a conjunction of \( \frac{n(n-1)}{2} \) disequality constraints when `A` has `n` elements:

```plaintext
forall(i, j in index_set(A) where i < j)(A[i] != A[j])
```

### Examples

- `n`-queens problem: see Topic 1: Introduction.
- Photo problem: see Topic 2: Basic Modelling.

An `alldifferent_except_0(A)` constraint allows multiple occurrences of the special dummy value `0`. 
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The \textbf{nvalue} Predicate

\textbf{Definition (Pachet and Roy, 1999)}

An \texttt{nvalue}(m, A) constraint holds if and only if variable $m$ takes the number of distinct values taken by the elements of the 1d array $A$ of variables, say with indices $1 \ldots n$:

$$|\{A[1], \ldots, A[n]\}| = m$$

The expression \texttt{nvalue}(A) denotes the number of distinct values taken by the elements of the 1d array $A$ of variables.

If $|A| = n$, then \texttt{nvalue}(n, A) means \texttt{alldifferent}(A):

Always use the most specific available constraint predicate!

\textbf{Example}

Model 2 of the Warehouse Location Problem:

see Topic 6: Case Studies.
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The **global_cardinality** Predicate

**Definition (Régisn, 1996)**

A global_cardinality \((A, V, C)\) constraint holds iff each variable \(C[j]\) has the number of elements of the 1d array \(A\) of variables that take value \(V[j]\). Variants exist.

Its default definition in MiniZinc includes:

\[
\text{forall}(j \in \text{index_set}(V))(\text{count}(A, V[j]) = C[j])
\]

It means alldifferent \((A)\) if \(\text{dom}(C[j]) = \{0, 1\}\) for each \(j\) and \(V = \bigcup_{i=1}^{n} \text{dom}(A[i])\), when \(A\) has indices \(1..n\): Always use the most specific available predicate!

**Example**

Model of the Magic Series problem: see Topic 4: Modelling.
A Common Source of Inefficiency in Models

Example

The model snippet

```plaintext
constraint forall(j in 1..n)
  (count(A,V[j]) = C[j]);
```

should be reformulated, due to the shared 1d array A, into:

```plaintext
constraint global_cardinality(A,V,C);
```

by applying the default definition backwards:

- at worst, it will be applied forwards while flattening;
- at best, the invoked solver has better inference.

This advice holds for all other global-constraint predicates.
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The **element** Predicate

**Definition (Van Hentenryck and Carillon, 1988)**

An *element*(i, A, e) constraint, where:

- A is an array of variables,
- i is an integer variable, and
- e is a variable,

holds if and only if A[i] = e.

For better model readability, the *element* predicate should not be used, as the functional form A[ϕ] is allowed, even if ϕ is an integer expression involving at least one variable.

**Example (Job allocation at minimal salary cost)**

**Given** jobs `Jobs` and the salaries of work applicants `Apps`, **find** a work applicant for each job **such that** some constraints (on the qualifications of the work applicants for the jobs, on workload distribution, etc) are satisfied and the total salary cost is minimal:

1. `array[Apps] of int: Salary;`
2. `array[Jobs] of var Apps: Worker; % job j by Worker[j]`
3. `solve minimize sum(j in Jobs)(Salary[Worker[j]]);`
4. `constraint ...; % qualifications, workload, etc`

Line 3 is equivalent to the less readable formulation

```
array[Jobs] of var 0..max(Salary): Cost; % Cost[j] for job j
constraint forall(j in Jobs)
  (element(Worker[j],Salary,Cost[j]));
solve minimize sum(Cost);
```

We do not know at modelling time the worker of each job!
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The \texttt{bin\_packing\_load} Predicate

\textbf{Definition}

Let item $i$ have the given weight or volume $V[i]$. Let variable $B[i]$ denote the bin into which item $i$ is put. Let variable $L[b]$ denote the load of bin $b$. A \texttt{bin\_packing\_load}(L, B, V) constraint holds iff each $L[b]$ is the sum of the $V[i]$ where $B[i]$ equals $b$. Variant predicates exist.

\textbf{Example (Balanced academic curriculum problem)}

Given, for each course $c$ in Courses, a workload $W[c]$ and a set $Pre[c]$ of prerequisite courses, find a semester $Sem[c]$ in $1..n$ for each course $c$ in order to satisfy all the prerequisites under a balanced workload:

1. \texttt{constraint bin\_packing(sum(W) div n, Sem, W);}
2. \texttt{constraint forall(c in Courses, p in Pre[c])(Sem[p]<Sem[c]);}
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The **knapsack** Predicate

**Definition**

Let item type \( t \) have the given weight or volume \( V[t] \).
Let item type \( t \) have the given value or profit \( P[t] \).
Let the variable \( X[t] \) denote the number of items of type \( t \) that are put into a given knapsack.
Let the variables \( v \) and \( p \) respectively denote the total volume and total profit of what is in the knapsack.
Given \( n \) item types, a **knapsack** \((V, P, X, v, p)\) constraint holds iff \( \sum(t \text{ in } 1..n) (V[t] \times X[t]) = v \) and \( \sum(t \text{ in } 1..n) (P[t] \times X[t]) = p \).

**Example**

To model the **Knapsack Problem** for a knapsack of given capacity \( c \), add \( v \leq c \) and maximize \( p \).
Example (https://xkcd.com/287)

A simplified version of the Knapsack Problem, but still NP-complete.

1 array[1..6] of int: Cost = [215,275,335,355,420,580];
2 array[1..6] of int: Joy = [ 0, 0, 0, 0, 0, 0, 0];
3 array[1..6] of var 0..(1505 div min(Cost)): Amount;
4 constraint knapsack(Cost, Joy, Amount, 1505, 0);
5 solve satisfy;

See this interview for some interesting trivia.
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
Assume we want to schedule a set of tasks to be performed over a given period such that we have the earliest end.

**Definition**

A task $T_i$ is a triple $\langle S[i], D[i], R[i] \rangle$ of constants or variables, where:

- $S[i]$ is the starting time of task $T_i$
- $D[i]$ is the duration of task $T_i$
- $R[i]$ is the quantity of a global resource needed by $T_i$

Tasks may be run in parallel if the global resource suffices.

**Sample schedule with parallel tasks and bounded resource**
**Definition**

A **precedence constraint** of task $T_1$ on task $T_2$ expresses that the performing of $T_1$ must finish **before** $T_2$ can start. We say that task $T_1$ **precedes** task $T_2$.

**Example (courtesy Magnus Ågren)**

Sample tasks (bubbles), durations (black numbers), resource requirements (blue numbers), and precedences (orange arrows). Task T7 is a dummy task, as we do not know which of tasks T5 and T6 will finish last.
Let us temporarily ignore the bounded global resource: If we have an unlimited global resource or each task has its own local resource, then the polynomial-time-solvable problem of finding the earliest ending time, under only the precedence constraints, for performing all the tasks can be modelled using linear inequalities.

Example (continued)

The precedence constraints indicated by the orange arrows on slide 24 are modelled as follows, based on the task durations indicated there in black:

```plaintext
1 constraint D = [2,1,4,2,3,1,0];
6 % add here the resource constraint of the next slide
7 solve minimize S[7];
```
The **cumulative** Predicate

### Definition (Aggoun and Beldiceanu, 1993)

A `cumulative(S,D,R,u)` constraint, where each task $T_i$ has a starting time $S[i]$, a duration $D[i]$, and a resource requirement $R[i]$, holds if and only if the resource upper limit $u$ is never exceeded when performing the $T_i$.

`cumulative` does not ensure any precedence constraints between the tasks: these have to be stated separately.

### Example (end)

To ensure that the global resource capacity of $u = 8$ units, say, is never exceeded under the resource requirements of the tasks indicated in blue on slide 24, add the following:

```constraint cumulative(S,D,[1,3,3,2,4,6,0],8);```
The **disjunctive** Predicate

**Definition**

A non-overlap constraint between tasks $T_1$ and $T_2$ states that either $T_1$ precedes $T_2$ or $T_2$ precedes $T_1$, say because both tasks require a resource that is available only for one task at a time. We say that tasks $T_1$ and $T_2$ do not overlap.

**Definition (Carlier, 1982)**

A disjunctive $(S, D)$ constraint, where each task $T_i$ has a starting time $S[i]$ and a duration $D[i]$, holds iff no two tasks $T_i$ and $T_j$ overlap in time. It is also known as unary.

It has the following definitions:

- **forall** $(i, j \text{ in } 1..n \text{ where } i<j)$
  
  $((S[i]+D[i] \leq S[j]) \lor (S[j]+D[j] \leq S[i]))$

- **cumulative** $(S, D, [1 \mid i \text{ in } 1..n], 1)$

Always use the most specific available constraint predicate!
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_leq
11. regular, table
12. Modelling Checklist
Enabling the representation of a circuit in a digraph:

- Let variable $S[v]$ represent the successor of vertex $v$.
- The domain of $S[v]$ is the set of vertices to which there is an arc from vertex $v$, plus $v$ itself.

**Example**

Assume the successor variables in $S$ take these values:

- $[b, c, d, a]$: one circuit $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$
- $[c, a, b, d]$: one subcircuit $a \rightarrow c \rightarrow b \rightarrow a$ and $S[d]=d$
- $[a, b, c, d]$: one empty subcircuit: $S[v]=v$ for all $v$ in Vertices
- $[c, d, a, b]$: two subcircuits, namely $a \rightarrow c \rightarrow a$ and $b \rightarrow d \rightarrow b$
- $[b, d, a, d]: c \rightarrow a \rightarrow b \rightarrow d$ is not a (sub)circuit
The **circuit** and **subcircuit** Predicates

**Definition (Laurière’78; Beldiceanu & Contejean’94)**

A **circuit** \( (S) \) constraint holds iff the arcs \( v \rightarrow S[v] \) form a Hamiltonian circuit: each vertex is visited exactly once.

A **subcircuit** \( (S) \) constraint holds iff \( \text{circuit}(S) \) holds for exactly one possibly empty but non-singleton subarray \( S' \) of \( S \), and \( S[v] = v \) for all the other vertices.

**Examples (Vehicle routing)**

Travelling salesperson problem (generalise this for vehicle routing problems with multiple vehicles or side constraints):

```c
3 solve minimize sum(c in Cities)(Dist[c,Next[c]]);
4 constraint circuit(Next);
```

Requiring a directed path from vertex \( v \) to vertex \( w \):

```c
constraint subcircuit(S) \( \setminus S[w] = v; \)
```

upon adding \( v \) to the domain of \( S[w] \) if need be.

Many graph constraints, including \texttt{dpath} exist in MiniZinc.
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
The **lex_lesseq** Predicate

**Example**

```plaintext
lex_lesseq([1,2,34,5,678], [1,2,36,45,78]),
because 34 < 36, even though 678 \nsubseteq 78.
```

**Definition**

A **`lex_lesseq(A,B)`** constraint, where `A` and `B` are same-length 1d arrays of variables, say both with indices in `1..n`, holds iff `A` is lexicographically at most equal to `B`:

- either `n=0`, or `A[1]<B[1]`,

Variant predicates exist.

**Usage:** Exploit index symmetries in matrix models, where there are arrays of variables: see Topic 4: Modelling, and see Topic 5: Symmetry.
Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_leq
11. regular, table
12. Modelling Checklist
### Regular Expressions

#### Examples (Regular Expressions)

- \((0|1)^*0\) denotes the set of even binary numbers.
- \(1^*(011^*)^*(0|\epsilon)\) denotes the set of strings of zeros and ones without consecutive zeros.
- \((0|1)^*00(0|1)^*\) denotes the set of strings of zeros and ones with consecutive zeros.

#### Notation for strings:

- Let \(\epsilon\) denote the empty string.
- Let \(v \cdot w\) denote the concatenation of strings \(v\) and \(w\).
- Let \(w^i\) denote the concatenation of \(i\) copies of string \(w\).
Definition

Let $\Sigma$ be an alphabet, that is a finite set of symbols. Regular expressions over $\Sigma$ are defined as follows:

- $\emptyset$ is a regular expression: its language, $L(\emptyset)$, is $\emptyset$.
- $\epsilon$ is a regular expression: $L(\epsilon) = \{\epsilon\}$.
- If $\sigma \in \Sigma$, then $\sigma$ is a regular expression: $L(\sigma) = \{\sigma\}$.
- If $r$ and $s$ are regular expressions, then $rs$ is a regular expression: $L(rs) = \{v \cdot w \mid v \in L(r) \land w \in L(s)\}$.
- If $r$ and $s$ are regular expressions, then $r|s$ is a regular expression: $L(r|s) = L(r) \cup L(s)$.
- If $r$ is a regular expression, then $r^*$ is a regular expression: $L(r^*) = \{w^i \mid i \in \mathbb{N} \land w \in L(r)\}$.

A regular expression defines a regular language over $\Sigma$. 

COCP / M4CO
Regular Expressions

Common abbreviations for regular expressions:
Let \( r \) be a regular expression:
- \( r^? \) denotes \( r | \varepsilon \); example in MiniZinc syntax: "12?"
- \( r^+ \) denotes \( rr^* \); example in MiniZinc syntax: "34+
- \( r^4 \) denotes \( rrrr \); example in MiniZinc syntax: "56{4}"
- \([1\ 2\ 3\ 4]\) denotes 1|2|3|4; same syntax in MiniZinc
- \([5\-8]\) denotes [5 6 7 8]; same syntax in MiniZinc
- \([9\-11\ 14]\) denotes [9 10 11 14]; same in MiniZinc
- \ldots \) (see the MiniZinc documentation)

Usage: Regular expressions are good for the specification of regular languages, but not so good for reasoning on them, where one often uses finite automata instead.
Deterministic Finite Automaton (DFA)

Example (DFA for regular expression \( ss(ts)^* | ts(tss)^* \))

Conventions:
- **Start state**, marked by arc coming in from nowhere: A.
- **Accepting states**, marked by double circles: D and E.
- **Determinism**: There is one outgoing arc per symbol in alphabet \( \Sigma = \{s, t\} \); missing arcs go to a non-accepting missing state that has self-loops on every symbol in \( \Sigma \).
The **regular** Predicate

**Definition (Pesant, 2004)**

A \texttt{regular}(A,Q,S,d,q0,F) constraint holds iff the values taken by the variables of the 1d array \texttt{A} form a string of the regular language accepted by the DFA with states 1..Q, symbols 1..S, transition function \texttt{d} in 1..Q x 1..S \rightarrow 0..Q with missing state 0, start state \texttt{q0}, and accepting states \texttt{F}. A \texttt{regular}(A,r) constraint holds iff \texttt{A} forms a string of the regular language denoted by the regular expression \texttt{r}.

**Example**

The DFA of the previous slide is represented as follows upon encoding the states \{A,B,C,D,E\} as 1..Q and the alphabet \{s,t\} as 1..S: we have Q=5 states, S=2 symbols, transition function \texttt{d}=[|2,3|4,0|5,0|0,2|3,5|], start state \texttt{q0}=1, and accepting states \texttt{F}={4,5}.
The table Predicate

Definition

A \text{table}(A, T) constraint holds iff the values taken by the 1d variable array $A$ form a row of the 2d value array $T$.

The 2d array $T$ gives an \textit{extensional definition} of a new constraint predicate, as opposed to the \textit{intensional definition} given so far for all other constraint predicates.

Example

If the variable array, say $X$, of the \textit{regular}(...) constraint of the previous slide for the DFA of two slides ago has four variables, then that constraint is equivalent to \text{table}(X, [ | 1,1,2,1 | 2,1,1,1 | 2,1,2,2 | ]).
Example (The Nonogram Puzzle: instance)

Each hint gives the sequence of lengths of blue blocks in its row or column, with at least one white cell between blocks, but possibly none before the first and after the last block.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (The Nonogram Puzzle: instance)

Each hint gives the sequence of lengths of **blue** blocks in its row or column, with at least one white cell between blocks, but possibly none before the first and after the last block.

Solution:
Example (The Nonogram Puzzle: model)

**Model:**

- **Variables:** An enumeration-type variable for each cell, with value $w$ if it is to be coloured white, and value $b$ if it is to be coloured blue.

- **Constraints:** State a `regular` constraint for each hint. For example, for a hint 2 3 1 on a row or column $A$ of length $n \geq 8$, state the constraint

  \[
  \text{regular}(A, \ "w* \ b{2} \ w+ \ b{3} \ w+ \ b{1} \ w*").
  \]

See **Survey of Paint-by-Number Puzzle Solvers:** the straightforward model above fares well, at least with a CP solver, compared to hand-written problem-specific code.
Example (Nurse Rostering)

Each nurse is assigned each day to one of the following:
- **N**  normal shift (this value is not available on Sundays)
- **L**  long shift (this value is not available on Sundays)
- **S**  Sunday shift (this value is only available on Sundays)
- **O**  day off

The nurse labour union imposes the following regulations:
- Monday off after a Sunday shift
- No single long shifts
- One day off after two consecutive long shifts

For each nurse \( n \), state the following constraint over the scheduling horizon, say 17 weeks here:

\[
\text{regular}(\text{Roster}[n, \text{Sun1..Sat17}], "(S O | L L O | N | O)^*")
\]

Further, a hospital has constraints on nurse presence.
Example (The Kakuro Puzzle: instance)

Fill in digits of 1 . . . 9 such that the digits of each word are pairwise distinct and add up to the number to the left (for horizontal words) or on top (for vertical words) of the word.
Example (The Kakuro Puzzle: instance)

Fill in digits of $1 \ldots 9$ such that the digits of each word are pairwise distinct and add up to the number to the left (for horizontal words) or on top (for vertical words) of the word.

```
   11  4
  5  2  3  10
14  2  3
   17
  9  5  1  2  3
  6  1  4  3  1
   10
  3  1  4  2
   3
  2  1
```
Example (The Kakuro Puzzle: first model)

**Model:**

- **Variables:** An integer variable for each cell, with domain $1..9$.

- **Constraints:** For each hint $K[\alpha] + \cdots + K[\beta] = \sigma$, state `alldifferent(i in \alpha..\beta)(K[i]) /
sum(i in \alpha..\beta)(K[i]) = \sigma`.

**Performance, using a CP solver:**

- $22 \times 14$ Kakuro with 114 hints: 9638 nodes, 160 s
- $90 \times 124$ Kakuro with 4558 hints: ? nodes, ? years

**Symptom:** The decomposition may give weak inference: for $x \neq y /\ \ x+y=4$, CP inference gives $x, y \ in \ 1..3$, not noticing that 2 should be pruned from both domains. We may need a custom predicate `alldifferent_sum`, constraining up to 9 variables over the domain $1..9$. 


Example (The Kakuro Puzzle: second model)

**New model:** Use the `regular` or `table` predicate for the `alldifferent` and `sum`-based constraints of each hint?

- For the hint $x+y=4$: `regular([x, y], "13|31").`
- For the hint $y+z=3$: `regular([y, z], "12|21").`
- One can also use `table` instead:
  ```
  table([x, y], [ | 1, 3 | 3, 1 | ]) \ /
  table([y, z], [ | 1, 2 | 2, 1 | ]).
  ```
- What about the hint $K[\alpha] + \cdots + K[\alpha+8] = 45$? There are $9! = 362,880$ solutions...
Example (The Kakuro Puzzle: second model, end)

New model (end):

- For the hint $K[\alpha] + \cdots + K[\alpha+8] = 45$, it suffices to state `alldifferent(i in \alpha..\alpha+8)(K[i])`, as the sum of 9 distinct non-0 digits is necessarily 45.
- For the hint $K[\alpha] + \cdots + K[\alpha+7] = \sigma$, it suffices to state `alldifferent([K[i] | i in \alpha..\alpha+7]++[45-\sigma])`.
- For the hint $K[\alpha] = \sigma$, it suffices to state $K[\alpha] = \sigma$.

Other opportunities for improvement exist.

New performance, using a CP solver:

- 22 × 14 Kakuro with 114 hints: 0 search nodes, 28 ms!
- 90 × 124 Kakuro with 4558 hints: 0 nodes, 345 ms!

Published diabolically hard Kakuros (like the 22 × 14 one mentioned above) where the new model pays off are rare.

The Kakuro story is based on material by Christian Schulte.
When to Use These Predicates?

Rapid prototyping of a new constraint predicate:
The `regular` and `table` predicates are very useful in the following conjunctive situation:

- A needed constraint predicate $\gamma$ on a 1d array of variables is not a built-in of MiniZinc or the used solver.
- A definition of $\gamma$ in terms of built-in predicates is not obvious to the modeller, or it has turned out that its inference is too expensive or too weak.
- The modeller does not have the time or skill to design an inference algorithm for $\gamma$, or deems $\gamma$ not reusable.
- The complexity and strength of an inference algorithm for $\gamma$ are not deemed crucial for the time being.
Important Modelling Device

Example (Encoding a small function)

The constraint \( x \times x = y \), where there is exactly one \( y \) for every \( x \), may yield poor inference: for small \( \text{dom}(x) \) try \text{element}(x, [i*i \mid i \in \text{dom}(x)], y), that is \([i*i \mid i \in \text{dom}(x)][x] = y\), for better inference.

The \text{element} predicate is a specialisation of \text{regular} and \text{table}, just like a function is a special case of a relation.

Example (Encoding a small relation)

The constraint \( x \times x = \text{abs}(y) \), where there can be more than one \( y \) for every \( x \), and vice-versa, may yield poor inference: for \( x \) in \( 0..3 \), say, try the less readable \( \text{table}([x,y], [\mid 0,0|1,-1|1,1|2,-4|2,4|3,-9|3,9\mid]) \) for better inference (maybe not with a MIP solver).
Bibliography


Outline

1. Motivation
2. alldifferent
3. nvalue
4. global_cardinality
5. element
6. bin_packing
7. knapsack
8. cumulative, disjunctive
9. circuit, subcircuit
10. lex_lesseq
11. regular, table
12. Modelling Checklist
Checklist for Designing or Reading a Model

- Predicates with most specific semantics are used
- Predicates with most global semantics are used
- The `element` predicate is not used explicitly, for clarity
- Functions are encoded if needed by implicit `element`
- Relations are encoded if needed by `regular` / `table`