LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

MiniZinc

KRZYSZTOF KUCHCINSKI
DEPT. OF COMPUTER SCIENCE, LTH

Outline

Introduction

MiniZinc

- Language for modelling constraint satisfaction and optimisation problems
  `http://www.minizinc.org`
- Mathematical-like notation syntax (quantifiers, coercion, overloading, sets, arrays)
- Expressive constraints (finite domain, set, (non-)linear arithmetic, integer)
- Strongly typed language
- Solver independent- can be used as front-end for many solvers (exist compiler to Flatzinc)
- Extensible (user-defined functions and predicates)
- Separation of data from model

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;
```

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;

predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
```

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;

predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;

constraint
   forall (i in 1..n, j in i+1..n)
              ( noattack(i, j, q[i], q[j]) );
```

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;

predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;

constraint
   forall (i in 1..n, j in i+1..n)
              ( noattack(i, j, q[i], q[j]) );

solve satisfy;
```

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;

predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;

constraint
   forall (i in 1..n, j in i+1..n)
              ( noattack(i, j, q[i], q[j]) );

solve satisfy;

output["q = \(q)"];
```

MiniZinc example

```
include "globals.mzn";

int: n;
array [1..n] of var 1..n: q;

predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;

constraint
   forall (i in 1..n, j in i+1..n)
              ( noattack(i, j, q[i], q[j]) );

solve satisfy;

output["q = \(q)"];

n = 8;
```

FlatZinc example

```
int: n = 8;

array[1 .. 8] of var 1 .. 8: q::output_array([ 1 .. 8 ]);

constraint int_lin_ne([ 1, -1 ], [ q[1], q[2] ], 1);

constraint int_ne(q[1], q[2]);

constraint int_lin_ne([ 1, -1 ], [ q[1], q[2] ], -1);

...
```

MiniZinc compilation

```
minizinc -c -Gjacop model.mzn
```

MiniZinc compilation

```
minizinc -c -Gjacop model.mzn

java -cp .:path_to_JaCoP org.jacop.fz.Fz2jacop [options] model.fzn
```

MiniZinc compilation

```
minizinc -c -Gjacop model.mzn

java -cp .:path_to_JaCoP org.jacop.fz.Fz2jacop [options] model.fzn
```

```
Usage: java org.jacop.fz.Fz2jacop [<options>] <file>.fzn
Options:
    -h, --help
        Print this message.
    -a, --all, --all-solutions
    -v, --verbose
    -t <value>, --time-out <value>
        <value> - time in milisecond.
    -s, --statistics
    -n <value>, --num-solutions <value>
        <value> - limit on solution number.
    -b, --bound - use bounds consistency whenever possible;
        overrides annotation ":: domain" and selects constraints
        implementing bounds consistency (default false).
    -sat use SAT solver for boolean constraints.
    -cs, --complementary-search - gathers all model, non-defined
         variables to create the final search
    -i, --interval print intervals instead of values for floating variables
    -p <value>, --precision <value> defines precision for floating operations
        overrides precision definition in search annotation.
    -f <value>, --format <value> defines format (number digits after decimal point)
        for floating variables.
    -o, --outputfile defines file for solver output
    -d, --decay decay factor for accumulated failure count (afc)
        and activity-based variable selection heuristic
    --step <value> distance step for cost function for floating-point optimization
```

MiniZinc compilation- cont'd

- Data in separate file (model.dzn)

```
minizinc -c -Gjacop model.mzn -d model.dzn
```

- For more options see

```
minizinc -h
```

MiniZinc compilation- cont'd

- Data in separate file (model.dzn)

```
minizinc -c -Gjacop model.mzn -d model.dzn
```

- For more options see

```
minizinc -h
```

- calling minizinc directly

```
minizinc --solver jacop -s model.mzn -d model.dzn
```

- For more options see

```
minizinc -h
```

```
minizinc -h jacop
```

Language elements

MiniZinc program

include statement

```
include "globals.mzn";
```

- includes other minizinc programs, data, libraries, etc.
- `globals.mzn` contains definition of global constraints;
  if compiled with `-G jacop` or `--solver jacop` includes JaCoP specific global constraints.

MiniZinc program

include statement

```
include "globals.mzn";
```

- includes other minizinc programs, data, libraries, etc.
- `globals.mzn` contains definition of global constraints;
  if compiled with `-G jacop` or `--solver jacop` includes JaCoP specific global constraints.

non-standard MiniZinc constraints and annotations (JaCoP specific)

```
include "jacop.mzn";
```

MiniZinc variables and parameter

Definition of variables and parameters

```
int: n;
var int: x;
array [1..n] of var 1..n: q;
```

MiniZinc variables and parameter

Definition of variables and parameters

```
int: n;
var int: x;
array [1..n] of var 1..n: q;
```

- **int:** n is a parameter known during compile time.
- **var int:** v is a finite domain variable.

MiniZinc variables and parameter

Definition of variables and parameters

```
int: n;
var int: x;
array [1..n] of var 1..n: q;
```

- **int:** n is a parameter known during compile time.
- **var int:** v is a finite domain variable.
- other examples

```
set of int: s;
var 1..n: q;
var {1, 5, 10}: f;

var bool: b;
```

MiniZinc literals

```
array[1..N, 1..N] of int: G;
...
N = 3;
G = [|1, -1,  0,
     |0,  2, -1,
     |2,  0, -1|];
```

MiniZinc literals

```
array[1..N, 1..N] of int: G;
...
N = 3;
G = [|1, -1,  0,
     |0,  2, -1,
     |2,  0, -1|];
```

or

```
N = 3;
G = array2d(1..N, 1..N, [
%   1   2  3
    1, -1,  0,  % 1
    0,  2, -1,  % 2
    2,  0, -1,  % 3
 ]);
```

MiniZinc literals (cont'd)

```
array[0..N-1, 0..N-1] of int: G;
```

MiniZinc literals (cont'd)

```
array[0..N-1, 0..N-1] of int: G;

N = 3;
G = array2d(0..N-1, 0..N-1, [
%   0   1  2
    1, -1,  0,  % 0
    0,  2, -1,  % 1
    2,  0, -1,  % 2
 ]);
```

MiniZinc enums

```
enum articles = {beef, bun, chees, onions, pickles, lettuce, ketchup, tomato};

array[articles] of int: calories;

array[articles] of var 1..5: quantity;

constraint
        sum(i in articles) (calories[i]*quantity[i]) < 3000
```

Array/set comprehension

```
int: z=10;
array [0..z] of 0..z*z: sq = array1d(0..z, [x*x | x in 0..z]);
```

Array/set comprehension

```
int: z=10;
array [0..z] of 0..z*z: sq = array1d(0..z, [x*x | x in 0..z]);

set of int: squares = {i * i | i in 1..(n * n) where i * i <= 5};

array [1..2*n] of var int: x = [c[i,j] | i in 1..n, j in 1..2];
```

Array/set comprehension

```
int: z=10;
array [0..z] of 0..z*z: sq = array1d(0..z, [x*x | x in 0..z]);

set of int: squares = {i * i | i in 1..(n * n) where i * i <= 5};

array [1..2*n] of var int: x = [c[i,j] | i in 1..n, j in 1..2];

bin_packing_load(cpu_loads, flow_processor,
                 [load[i, j] | i in 1..2, j in 1..no_flows])
```

Array/set comprehension

```
int: z=10;
array [0..z] of 0..z*z: sq = array1d(0..z, [x*x | x in 0..z]);

set of int: squares = {i * i | i in 1..(n * n) where i * i <= 5};

array [1..2*n] of var int: x = [c[i,j] | i in 1..n, j in 1..2];

bin_packing_load(cpu_loads, flow_processor,
                 [load[i, j] | i in 1..2, j in 1..no_flows])

forall (i in 1..graph_size)
        (next[i] in {to[j] | j in 1..n where from[j] = i} union
         {from[j] | j in 1..n where to[j] = i} union
         {i | j in 1..n where i != start} union % not self-loop for start
         {start | j in 1..n where dest = i}) % subcircuit closing
```

Predicates and Tests

Predicate

```
predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
```

Predicates and Tests

Predicate

```
predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
```

```
predicate decreasing(array[int] of var int: x) =
   forall(i in 2..length(x)) ( x[i-1] >= x[i]  );
```

Predicates and Tests

Predicate

```
predicate noattack(int: i, int: j, var int: qi, var int: qj) =
      qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
```

```
predicate decreasing(array[int] of var int: x) =
   forall(i in 2..length(x)) ( x[i-1] >= x[i]  );

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

- `test` only involve parameters,
- `test` can be used inside the test of a conditional expression.

Predicates- advanced example

```
predicate diff2(array[int,int] of var int: r) =

    let { int: lr2 = min(index_set_2of2(r)),
          int: ur2 = max(index_set_2of2(r)),
          int : s2 = (ur2 - lr2 + 1),
        }
        in
          assert( s2 = 4,
            "diff2: size of a rectangle must be 4",
             jacop_diff2(r)
          );

predicate jacop_diff2(array[int,int] of var int: r);
```

Functions

```
function var int: count(array[int] of var int: x, var int: y) =
  let {
     var 0..length(x): c;
     constraint count(x,y,c);
  } in c;
```

Functions

```
function var int: count(array[int] of var int: x, var int: y) =
  let {
     var 0..length(x): c;
     constraint count(x,y,c);
  } in c;
```

Example:

```
constraint
  c = count(x, v);
```

instead of

```
constraint
  count(x, v, c);
```

Constraints

Constraints

```
constraint
   forall (i in 1..n, j in i+1..n) ( noattack(i, j, q[i], q[j]) );
```

Constraints

Constraints

```
constraint
   forall (i in 1..n, j in i+1..n) ( noattack(i, j, q[i], q[j]) );
```

```
constraint
   x + y > z;

constraint
  x > 0 /\ x + y = z;

constraint
  x + 3 = z \/ a - b = c;
```

Constraints

Constraints

```
constraint
   forall (i in 1..n, j in i+1..n) ( noattack(i, j, q[i], q[j]) );
```

```
constraint
   x + y > z;
```

```
constraint
  x > 0 /\ x + y = z;
```

```
constraint
  x + 3 = z \/ a - b = c;
```

```
constraint
   (x > z) -> (y = 0);
```

```
constraint
  x > 0 <-> y = z;
```

```
constraint
  if c = 0 then d = a - b
           else c = 55 endif;
```

Constraints- cont'd

```
constraint
   forall (i in 1..N) (a[i] > 0);

constraint
  exists (i in 1..N) (a[i] = 0);

constraint
   x = sum(i in 1..n)(a[i]);
```

Constraints- cont'd

```
constraint
   forall (i in 1..N) (a[i] > 0);

constraint
  exists (i in 1..N) (a[i] = 0);

constraint
   x = sum(i in 1..n)(a[i]);

constraint
    circuit(x) /\
    forall(i in 1..n) (distances[i, x[i]] = d[i])
    /\
   distance = sum(i in 1..n) (d[i]);
```

Constraints- cont'd

```
constraint
   forall (i in 1..N) (a[i] > 0);

constraint
  exists (i in 1..N) (a[i] = 0);

constraint
   x = sum(i in 1..n)(a[i]);

constraint
    circuit(x) /\
    forall(i in 1..n) (distances[i, x[i]] = d[i])
    /\
   distance = sum(i in 1..n) (d[i]);

constraint sum (i in n) ((b[i]>0)*w[i])
```

Solve statement

Solve

```
solve satisfy;
```

Solve statement

Solve

```
solve satisfy;
```

```
solve satisfy;

solve minimize x;

solve maximize x;

solve minimize sum(i in 1..N)(x[i]);
```

Solve annotations

- define variables for search.
- define search parameters, such as variable selection, value selection.

```
solve :: int_search(x, input_order, indomain_min, complete)
                  satisfy;

solve :: bool_search(b, input_order, indomain_min, complete)
                   satisfy;

solve :: set_search([s], input_order, indomain_min, complete)
                   maximize c;
```

Solve annotations (cont'd)

```
solve :: int_search([crew[f,p] | f in 1..numFlights,
                     p in 1..numPersons ],
            first_fail, indomain, credit(5, bbs(5))) minimize z;
```

```
solve :: seq_search([
      int_search(t, smallest, indomain_min, complete),
      int_search(r, input_order, indomain_min, complete)
]) minimize end;
```

Solve annotations- JaCoP specific

```
solve :: priority_search(distance_to_deadline,
          [seq_search([int_search([start[i]], input_order, indomain_min),
            int_search([machine[i]], input_order, indomain_random)]) | i in 1..n],
           smallest) minimize cost;
```

Solve annotations- JaCoP specific

```
solve :: priority_search(distance_to_deadline,
          [seq_search([int_search([start[i]], input_order, indomain_min),
            int_search([machine[i]], input_order, indomain_random)]) | i in 1..n],
           smallest) minimize cost;

solve :: restart_luby(n*50)
      :: priority_search(distance_to_deadline,
          [seq_search([int_search([start[i]], input_order, indomain_min),
            int_search([machine[i]], input_order, indomain_random)]) | i in 1..n],
           smallest) minimize cost;
```

Output statement

Output

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

Output statement

Output

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

```
output [show(x), show(y)];
```

- **output** statement is more complicated and makes it possible to specify formatting;
- flatzinc compiles this to ozn file that can be used with command `solns2out`

```
fzn-jacop -s x.fzn | solns2out x.ozn
```

Output statement example

```
output["Voucher paid:\n"]++
[if j mod n != 0 then show(voucher_paid[i,j]) ++ "  "
                 else show(voucher_paid[i,j]) ++ "\n" endif
  | i in 1..m, j in 1..n]
```

Output statement example

```
output["Voucher paid:\n"]++
[if j mod n != 0 then show(voucher_paid[i,j]) ++ "  "
                 else show(voucher_paid[i,j]) ++ "\n" endif
  | i in 1..m, j in 1..n]
```

or

```
output["Voucher paid:\n"]++
[if j mod n != 0 then "\(voucher_paid[i,j])  "
                 else "\(voucher_paid[i,j])\n" endif
  | i in 1..m, j in 1..n]
```

Output statement example

```
output["Voucher paid:\n"]++
[if j mod n != 0 then show(voucher_paid[i,j]) ++ "  "
                 else show(voucher_paid[i,j]) ++ "\n" endif
  | i in 1..m, j in 1..n]
```

or

```
output["Voucher paid:\n"]++
[if j mod n != 0 then "\(voucher_paid[i,j])  "
                 else "\(voucher_paid[i,j])\n" endif
  | i in 1..m, j in 1..n]
```

```
Voucher paid:
0  0  0  0  0  0  1  0  0  0
0  0  1  1  0  0  0  0  0  0
0  0  0  0  0  1  0  0  0  0
0  1  0  0  1  0  0  0  0  1
```

Trace

- the function `trace(s,e)` prints the string `s` before evaluating the expression `e` and returning its value.
- it can be used in **any context**.
- example

```
constraint
   forall (i in 1..n, j in i+1..n)
      (trace("generate for i=\(i) and j=\(j)\n",
             noattack(i, j, q[i], q[j])))
;
```

Example

Larger example- perfect square

```
include "globals.mzn";

int: n; % number squares
int: size; % size of the square to fill with n squares
array[1..n] of int: squares; % dimension of the squares

array[1..n] of var 0..size: x; % x position of squares
array[1..n] of var 0..size: y; % y position of squares

constraint
   forall(i in 1..n) (
      x[i] <= size - squares[i] /\
      y[i] <= size - squares[i] ) /\
   diffn(x, y, squares, squares);

solve :: seq_search([
      int_search(x, smallest, indomain_min, complete),
      int_search(y, smallest, indomain_min, complete)
      ]) satisfy;

output[show(x)] ++ ["\n"] ++ [show(y)]

n=21;
size=112;
squares = [2,4,6,7,8,9,11,15,16,17,18,19,24,25,27,29,33,35,37,42,50];
```

Result

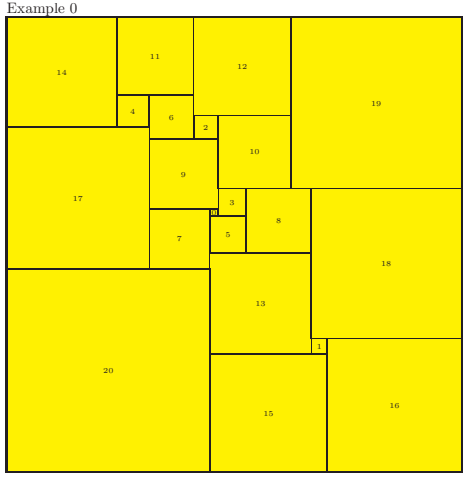
```
> fzn-jacop -s perfect_square.fzn
x = array1d(1..21, [50, 75, 46, 52, 27, 50, 35, 35, 59, 35, 52, 27, 46, 50, 0, 50, 79, 0, 75, 70, 0]);
y = array1d(1..21, [63, 29, 82, 63, 85, 54, 82, 50, 54, 65, 70, 93, 88, 29, 85, 0, 0, 50, 33, 70, 0]);
----------

%% Model variables : 86
%% Model constraints : 1

%% Search CPU time : 258ms
%% Search nodes : 430
%% Search decisions : 225
%% Wrong search decisions : 205
%% Search backtracks : 188
%% Max search depth : 39
%% Number solutions : 1

%% Total CPU time : 353ms
```

Result



Conclusions

Conclusions

- Convenient language for specification of constraint satisfaction and optimization problems.
- Mathematical-like notation syntax.
- Predicates make possible constraint decomposition.
- Solver independent.

LUND
UNIVERSITY

LTH

FACULTY OF
ENGINEERING