“We heard what you said but we knew what you meant”

Automatic Reformulation in MiniZinc

Peter J. Stuckey

Overview

- A little bit about MiniZinc
    - Predicates, functions, and flattening
- Automatic Reformulations
    - Linearization, Sets, and Strings
    - Multi pass compilation
    - Autotabling
    - Symmetry detection
    - Globals detection
- Conclusion

“Alone we can do so little; together we can do so much.” – Helen Keller

Roberto Amidini, Maria Garcia de la Banda, Gustav Bjordal, Jip Dekker, Thibaut Feydy, Pierre Flener, Graeme Gange, Tias Guns, David Hemmi, Kevin Leo, Kim Marriott, Chris Mears, Nick Nethercote, Justin Pearson, Andrea Rendl, Andreas Schutt, Joseph Scott, Guido Tack, Mark Wallace

MiniZinc Predicates

- MiniZinc is based on model rewriting
- Predicates: define a new (global) constraint

```
predicate alldifferent(array[int] of var int: x)
         = forall(i,j in index_set(x) where i < j)
                  (x[i] != x[j]);
```

- Essential to treatment of globals
  - solvers use a default decomposition, or
  - replace with their own decomposition or direct constraint

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

- Advantages: all globals available for all solvers

MiniZinc Functions

- Its also useful to have functions

```
function array[int] of var int: global_cardinality
         (array[int] of var int: x, array[int] of int: v)
= let { array[index_set(v)] of var int: c
         = [ sum(i in index_set(x))(x[i] = v[j])
           | j in index_set(v) ]; }
  in c;
```

- Common subexpression elimination is better
  - almost a third of the global constraint catalog are functions
- It also makes the MiniZinc core simpler

```
function var int: abs(var int: x) =
     let { int: m = max(-lb(x),ub(x));
           var -m..m: y;
           constraint int_abs(x,y); }
     in y;
```

local constraints

Flattening

- Mapping a high level model
  - complex loops
  - deep expressions
  - functions and predicates
- To a flat model
  - variables
  - constraints
  - objective

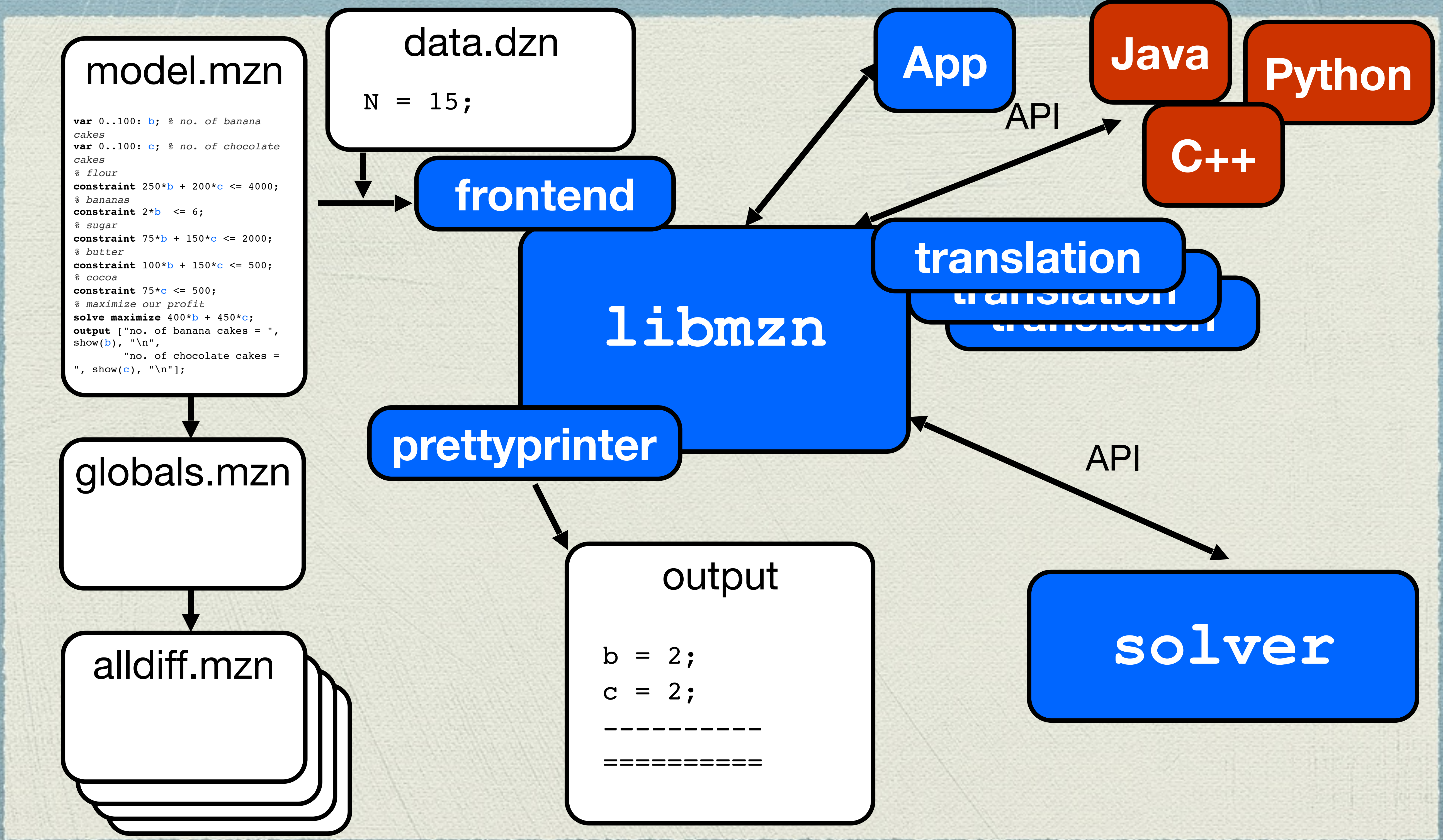
```
% (square) job shop scheduling in MiniZinc
int: size;                                    % size of problem
array [1..size,1..size] of int: d;            % task durations
int: total = sum(i,j in 1..size) (d[i,j]);  % total duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end;                            % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint
    forall(i in 1..size) (
        forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,size] + d[i,size] <= end /\
        forall(j,k in 1..size where j < k) (
            no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
        )
    );

solve minimize end;
```

```
array[0..3] of var 0..14: s;
var 0..14: end;
var bool: b1;
var bool: b2;
var bool: b3;
var bool: b4;
constraint  int_lin_le     ([1,-1], [s[0], s[1]], -2);
constraint  int_lin_le     ([1,-1], [s[2], s[3]], -3);
constraint  int_lin_le     ([1,-1], [s[1], end ], -5);
constraint  int_lin_le     ([1,-1], [s[3], end ], -4);
constraint  int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1);
constraint  int_lin_le_reif([1,-1], [s[2], s[0]], -3, b2);
constraint  bool_or(b1, b2, true);
constraint  int_lin_le_reif([1,-1], [s[1], s[3]], -5, b3);
constraint  int_lin_le_reif([1,-1], [s[3], s[1]], -4, b4);
constraint  bool_or(b3, b4, true);
solve minimize end;
```

Critical Flattening Steps

- All standard in language compilers
- Constant folding
- Common Subexpression Elimination
  - two names for the same thing is deadly for CP
  - particularly for learning solvers
- Equality tracking
  - substitution/elimination of common names

libmzn



```
var 0..100: b; % no. of banana
cakes
var 0..100: c; % no. of chocolate
cakes
% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b  <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;
% maximize our profit
solve maximize 400*b + 450*c;
output ["no. of banana cakes = ",
show(b), "\n",
        "no. of chocolate cakes =
", show(c), "\n"];
```

```
N = 15;
```

```
b = 2;
c = 2;
----------
==========
```

Automatic Reformulation

Sets

- MiniZinc mantra
  - your model runs on all solvers
- Problem: Set variables are not supported
- Solution `nosets.mzn` (200 lines of code)
  - translate set variables to arrays of booleans
    - crucial use of functions to avoid multiple translations
  - convert set operations to functions on arrays
  - no set variables in the final FlatZinc

Strings

- MiniZinc extended to include string variables
  - not yet released
- String solving not supported by most solvers
  - only Gecode+S
- Map strings to existing FlatZinc
  - Translate strings to arrays of integers
  - Map constraints on strings to constraints on arrays
  - Map string operations to operations on arrays
    - concatenation, reverse, length, regular, gcc, lexorder
  - Not that uncompetitive wrt to Gecode+S

Linearization

- The most important transformation
    - allows MiniZinc to run on MIP solvers
    - beware they are quite competitive on CP problems
- Linearization consists of
    - specialised linear global decompositions

```
predicate alldifferent(array[int] of var int: x)
 = forall(j in array_dom(x))
         (sum(i in index_set(x))(x[i] == j) <= 1);
```

- general linearization by "big M" methods
- special treatment of constraints on variables domains (`x in S`)
- NEW: some globals treated as separators, e.g. `circuit`

Multi Pass Compilation

- MiniZinc flattens to FlatZinc
  - many decisions made during flattening, e.g

```
var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
constraint x+y+z=12 -> y=max([x,y,z]);
```

  - becomes

```
var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
var 2..5: i0 = max([x,y,z])
var bool: b0 = (y = i0)
var bool: b1 = (x+y+z != 12)
constraint or(b0,b1);
```

Multi Pass Compilation

- More information = better decisions

```
var {2,4}: x; var {2,4}: y; var {~~2,4,~~5}: z;
constraint all_different([x,y,z]);
var 2..5: i0 = ~~max([x,y,z])~~  5
var bool: b0 = ~~(y = i0)~~       false
var bool: b1 = (x+y+~~z~~5 != 12) true
constraint ~~or(b0,b1);~~
```

- finally

```
var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
constraint x+y != 7;
```

Multi Pass Compilation

- Multi pass compilation
  - Key requirement: variable and expression paths
  - Gecode first pass: Other solver second pass
  - reduces model size: around 5%
  - reduces run time for MIP solvers: sometimes 50%
  - can improve compile time, no worse than double

Auto Tabling

- Annotate a predicate as: `:: presolve(autotable);`

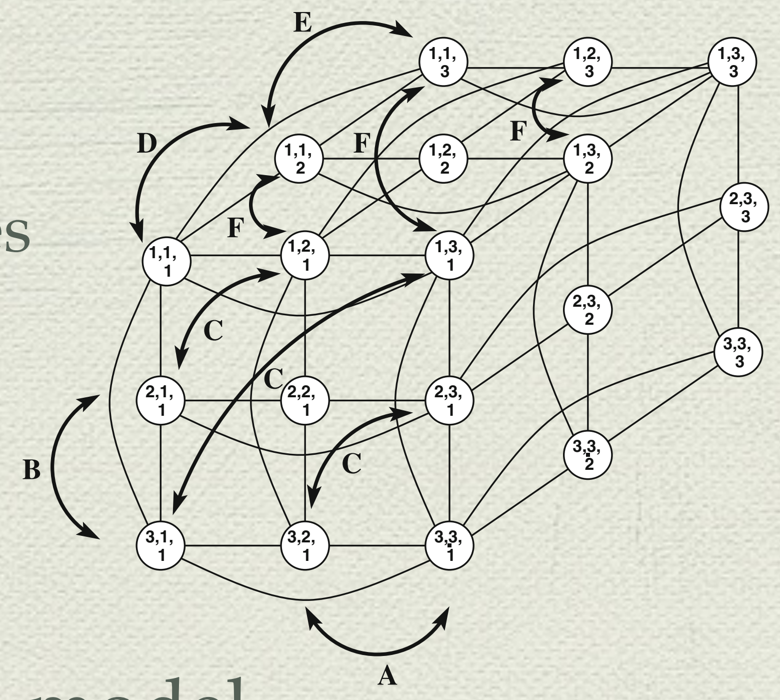
```
predicate rank_apart(var 1..52: a, var 1..52: b)
  = table(abs([a,b](a|-1b)2 mod1,133)|in {1,12};51 |]);
```

- Solution are computed
  - predicate replaced by a table constraint
- Variations
  - call-based, and instance independent
- Benefits
  - improved solving time
  - automatic reformulation of poor models
- Not done in Australia

Learning Reformulation

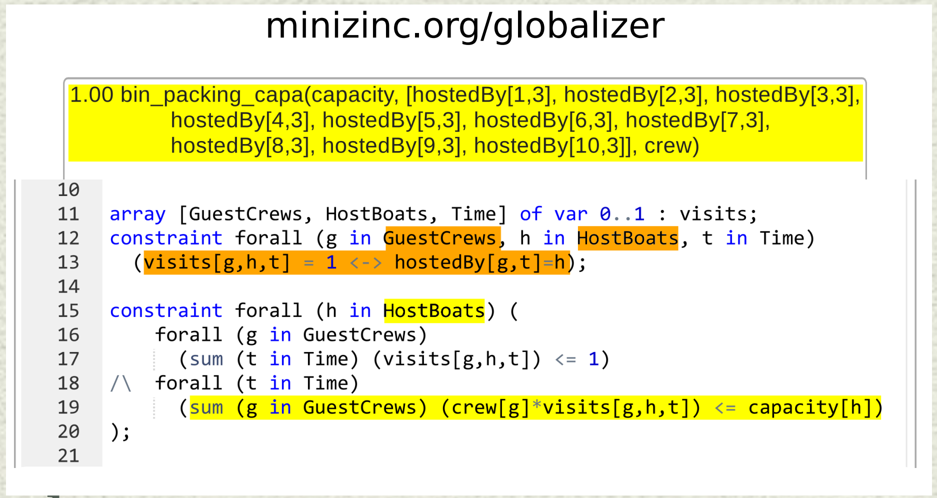
Symmetry Detection

- Generate symmetries of small instances
  - find which symmetries generalize across instances
- Generate candidate model symmetries
  - ask the user or use theorem proving
- Add symmetry breaking (dynamic/static) to model
- Extension to dominance
  - separate out objective and/or some constraints
  - generate symmetries
  - convert to dominance constraints



Globals Detection

- Find global constraints which are implied by the model
  - Use structure of model to find sub-problems
  - Generate candidate global constraints
  - Rank the global candidates by
    - coverage by solutions, size of global
  - Present the globals to the user in ranked order
- Was available as a web tool: minizinc.org/globalizer
- Highly important approach for non-expert modellers
  - gives a way to "lookup" the globals you need for your problems


minizinc.org/globalizer

```
1.00 bin_packing_capa(capacity, [hostedBy[1,3], hostedBy[2,3], hostedBy[3,3],
         hostedBy[4,3], hostedBy[5,3], hostedBy[6,3], hostedBy[7,3],
         hostedBy[8,3], hostedBy[9,3], hostedBy[10,3]], crew)
```

```

array [GuestCrews, HostBoats, Time] of var 0..1 : visits;
constraint forall (g in GuestCrews, h in HostBoats, t in Time)
  (visits[g,h,t] = 1 <-> hostedBy[g,t]=h);

constraint forall (h in HostBoats) (
    forall (g in GuestCrews)
      (sum (t in Time) (visits[g,h,t]) <= 1)
/\  forall (t in Time)
      (sum (g in GuestCrews) (crew[g]*visits[g,h,t]) <= capacity[h])
);

```

Other Reformulations

- Bounds versus Domain propagation
  - we can analyse models to determine that bounds propagators will fail at the same time as domain
- Multiple reformulations (model portfolios)
  - e.g. map sets to multiple representations: array of bool, array of int
  - Essence trys all possible reformulations
- Adding implied constraints
  - similar to symmetry and globaliser: which constraints to add
- Associative Commutative CSE
  - use AC matching to find more CSE
  - can be much better than normal CSE on the right examples

The Holy Grail

Conclusion

- MiniZinc is a modelling language based on reformulation
  - essential to supporting varied solvers (linearization)
- Automatic Reformulation is widely used
  - language extensions by reformulation (sets, strings)
  - improving model flattening (multi-pass, auto tabling)
  - recognizing ways to improve a model (symmetry + globals detection)
- Exciting new directions
  - "Learning from Learning Solvers" CP2016 showed we can improve our models by looking at how learning solvers solve them!
  - "Automatic LBBD solving" AAAI2017 how we can create a hybrid MIP/CP solution to any model that uses the strength of both

Progress to the Holy Grail

- Better modelling languages,
  - supported by automatic reformulation
  - is a critical step towards the holy grail
- CP is closer than it was, but we need it to
  - easier to learn
  - better analysis/transformation of models
  - faster solving
- Remember the Holy Grail is (at least in theory) unattainable
  - But that should not stop us reaching for it