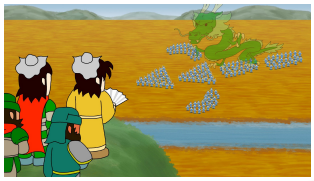
Contexts

Jimmy Lee & Peter Stuckey

The Azure Dragon Army Formation

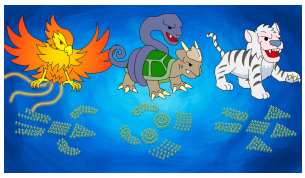


2

The Four Mythological Creatures

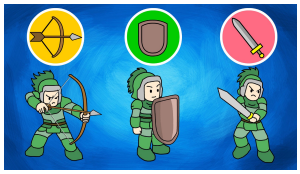
3

More Army Formations



4

Soldier Specialties

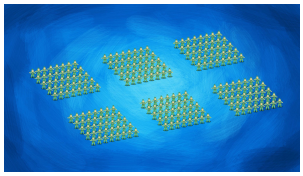


5

24 Battalions

6

6 Battalions = 1 Brigade

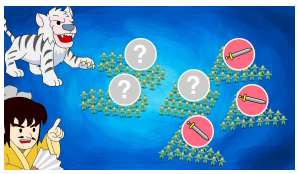


1

The Vermillion Bird Formation

2

The White Tiger Formation



3

The Black Turtle Formation

4

At Least Two Elite Brigades

11

Brigade Creation (brigades.mzn)

- Data

```
enum BATTALION;
set of BATTALION: ARCHER;
set of BATTALION: SWORD;
set of BATTALION: SHIELD;
int: size; % size of brigade
```

- Decisions + Size/disjoint constraints

```
int: nbrigade = card(BATTALION) div size;
set of int: BRIGADE = 1..nbrigade;
array[BRIGADE] of var set of BATTALION: S;
include "all_disjoint.mzn";
all_disjoint(S);  % disjoint array of sets
forall(i in BRIGADE)(card(S[i]) = size;
```

12

Brigade Creation (brigades.mzn)

- Elite brigade definition

```
predicate elite(var set of BATTALION: brigade) =
   pattern(brigade,[ARCHER,ARCHER,SHIELD,SHIELD])
\/ pattern(brigade,[SWORD,SWORD,SWORD])
\/ pattern(brigade,[ARCHER,SWORD,SWORD,SHIELD]);

include "alldifferent.mzn";
predicate pattern(var set of BATTALION: brigade,
            array[int] of set of BATTALION: t) =
   let { set of int: IND = 1..length(t);
         array[IND] of var BATTALION: b; } in
   forall(i in IND)
         (b[i] in t[i] /\ b[i] in brigade)
   /\ alldifferent(b);
```

13

Brigade Creation (brigades.mzn)

- Solving the satisfaction problem first …
  - First two brigades must be elite

```
constraint elite(S[1]) /\ elite(S[2]);
solve satisfy;
```

- What happens

```
MiniZinc: flattening error: 'all_different_int'
is used in a reified context but no reified
version is available
```

- What is going on?

14

Root Contexts

- Constraint solvers solve a conjunction of constraints!
- Models with disjunction, implication, bool2int, are translated to conjunction
- A constraint *c* appears in a root context iff
  - it appears as: `constraint` *c*
  - it appears as: (*c* `/\` ...) in a root context
  - it appears as: `forall`(..)(*c*) in a root context
- Alternatively
  - the only connectives between *c* and `constraint` are conjunction (`/\` and `forall`)

15

Root Contexts Example

- Root context for both elite calls

```
constraint elite(S[1]) /\ elite(S[2]);
```

- First constraint is flattened to

```
   pattern(S[1],[ARCHER,ARCHER,SHIELD,SHIELD])
\/ pattern(S[1],[SWORD,SWORD,SWORD])
\/ pattern(S[1],[ARCHER,SWORD,SWORD,SHIELD]);
```

- Pattern calls are not in a root context
- First pattern is flattened to

```
let { array[1..4] of var BATTALION: b; } in
   b[1] in ARCHER /\ b[1] in S[1] /\
   b[2] in ARCHER /\ b[2] in S[1] /\
   b[3] in SHIELD /\ b[3] in S[1] /\
   b[4] in SHIELD /\ b[4] in S[1] /\
   /\ alldifferent(b);
```

- Alldifferent call is not in root context

16

Reification

- Solvers take a conjunction of constraints
- So how are negation/disjunction treated
- Reification
  - attaching a Boolean to a constraint which is
    - true if the constraint holds
    - false if the constraint doesn't hold
- Example

```
constraint x > y \/ not p(x,y);
```

- Becomes

```
constraint b1 <-> x > y;
constraint b2 <-> p(x,y);
constraint b1 \/ not b2;
```

17

Globals in Non-Root Contexts

- Global constraints in non-root contexts are usually not supported, e.g.

```
(c \/ alldifferent(s))
```

- MiniZinc will try to use a new predicate
  - alldifferent_reif(s,b) ≡ b ↔ alldifferent(s)

```
(c \/ b)  /\ alldifferent_reif(s,b)
```

- Most solvers don't implement reified versions of global constraints
- "Default" definitions of MiniZinc globals in the library works with reification
  - slow since implemented by decomposition
- **Avoid** using globals in non-root contexts

18

Brigade Creation (brigades.mzn)

- Fix for problem
- Define our own alldifferent via decomposition

```
include "alldifferent.mzn";
predicate
alldifferent(array[int] of var BATTALION: b)
   = forall(i,j in index_set(b) where i < j)
          (b[i] != b[j]);
```

- Base constraints can be reified

19

Brigade Creation (brigades.mzn)

- The model now runs, and returns e.g

```
Brigade 1: {B1, B2, B9, B15, B17, B18}
Brigade 2: {B3, B6, B8, B13, B14, B16}
Brigade 3: {B19, B20, B21, B22, B23, B24}
Brigade 4: {B4, B5, B7, B10, B11, B12}
----------
```

- But we are trying to maximise elite brigades

```
constraint elite(S[1]) /\ elite(S[2]);
solve satisfy;
solve maximize sum(i in BRIGADE)(elite(S[i]))
```

- But now …

20

Brigade Creation (brigades.mzn)

- MiniZinc now returns:

```
MiniZinc: flattening error:
brigades.mzn:40:
  in call 'sum'
  in call 'bool2int'
  in array comprehension expression
    with i = 1
  in call 'elite'
brigades.mzn:31:
  in binary '\/' operator expression
brigades.mzn:32:
  in call 'pattern'
brigades.mzn:25:
  in let expression
  in variable declaration for 'b'
  free variable in non-positive context
```

- Yikes!

21

Positive, Negative + Mixed Contexts

- A constraint *c* is in a positive context (+) iff
  - there are an even number of negations on the path from `constraint` to the constraint *c*
  - root contexts are always positive
- A constraint under an odd number of negations is in a negative context (-)
- Some operators created mixed contexts (±)
  - `bool2int`(c): c in a mixed context
  - c1 <-> c2: both c1 and c2 in mixed context
- Mixed contexts are effectively under an odd and even number of negations

22

Positive, Negative + Mixed Contexts

- Count the negations from `constraint`
- Context propagation (downwards)
  - `constraint +` % <+ve context>
  - +: `not −`
  - −: `not +`
  - ±: `not ±`
  - <*cn*>: <*cn*> `\/` <*cn*> %<cn> is +, - or ±
  - <*cn*>: <*cn*> `/\` <*cn*>
  - treat A -> B as `not` A `\/` B
  - <*cn*>: ± <-> ±
  - <*cn*>: `bool2int(±)`

23

Uninitialized Variables and Contexts

- A let-in construct cannot introduce uninitialized variables, except in a positive context
- Example

```
predicate even( var int:x ) =
          let { var int: y } in x = 2*y;
var -1..7: u;
constraint not even(u);
```

- Result

```
MiniZinc: flattening error: even.mzn:4:
  in unary 'not' operator expression
  in call 'even'
even.mzn:2: in let expression
  in variable declaration for 'y'
  free variable in non-positive context
```

24

Uninitialized Variables and Contexts

- What is the problem?
  - We are trying to ask
    not even(u)
    ↔ not (let {var int: y} in u = 2*y)
    ↔ not (∃ y. u = 2*y)
    ↔ ∀ y (not u = 2 * y)
    ↔ ∀ y (u ≠ 2 * y)
- Constraint solvers do not support
  - universal quantification!

25

Back to Brigade Creation

- The call to `pattern` appears in a mixed context
  - pattern introduces uninitialized variables

```
let { set of int: IND = 1..length(t);
      array[IND] of var BATTALION: b; } in
```

- Why is it mixed?

```
solve maximize sum(i in BRIGADE)(elite(S[i]));
```

- There is an implicit `bool2int` inserted by the MiniZinc compiler

```
solve maximize sum(i in BRIGADE)
                  (bool2int(elite(S[i])));
```

26

Solving Brigade Creation

- Do the optimisation by hand

```
constraint elite(S[1]) /\ elite(S[2]) /\
           elite(S[3]);
solve satisfy;
=====UNSATISFIABLE=====
```

- So optimal value is 2!
- Or replace `var` by `exists`

```
predicate pattern(var set of BATTALION: brigade,
                  array[int] of set of BATTALION: t) =
   exists(b1 in t[1])(b1 in brigade /\
   exists(b2 in t[2])(b2 in brigade /\ b1 != b2 /\
   exists(b3 in t[3])(b3 in brigade /\ b1 != b3 /\ b2 != b3 /\
   if length(t) >= 4 then
      exists(b4 in t[4])(b4 in brigade /\ b1 != b4 /\
                         b2 != b4 /\ b3 != b4)
   else true endif)));
```

- Quite tricky and very very slow to solve!

27

Summary

- Contexts: root, positive, negative, mixed
  - context is determined but the path from the root expression to the subexpression of interest
- Globals can usually only appear in root contexts
- Local decisions without definition, i.e. uninitialized
  - can only occur in positive contexts

28

Image Credits

All graphics by Marti Wong, ©The Chinese University of Hong Kong and The University of Melbourne 2016