Trials and Tribulations of Designing a Modelling Language

Maria Garcia de la Banda, Kim Marriott, Nicholas Nethercote, Reza Refah, Peter J. Stuckey, Mark Wallace, Peter Baumgartner, Ralph Becket, Sebastian Brand, Mark Brown, Gregory J. Duck, Julien Fischer, Michael Maher, Jakob Puchinger, John Slaney, Zoltan Somogyi, Toby Walsh

September 21, 2007

Outline

1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. Natural Modelling
4. Extensible Modelling
5. Software Engineering
6. Practical Solver-Independence
7. Conclusion

Outline

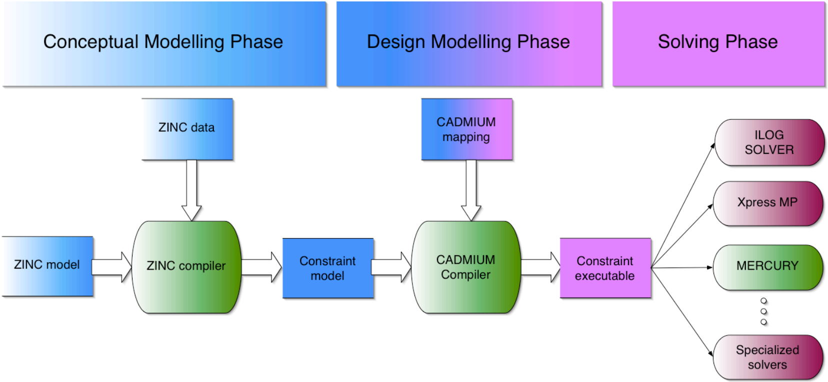
1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. Natural Modelling
4. Extensible Modelling
5. Software Engineering
6. Practical Solver-Independence
7. Conclusion

Problem Solving Process

- Problem
  - "Find 4 different integers between 1 and 5 that sum to 14"
- Conceptual Model
  - Precisely specify the problem without describing how to solve it
  - $S \subseteq \{1,2,3,4,5\} \wedge |S| = 4 \wedge \sum S = 14$
- Design Model
  - Correct efficient algorithm
  - Specified using some solver technology and search strategy
  - `[W,X,Y,Z] :: 1..5, alldifferent([W,X,Y,Z]), W + X + Y + Z #= 14, labeling([X,Y,Z,T])`
- Solution
  - $W = 2 \wedge X = 3 \wedge Y = 4 \wedge Z = 5 \Rightarrow S = \{2,3,4,5\}$

G12

- Designed to mimic the problem solving process
- Zinc: Conceptual modelling language
- Cadmium: Mapping langauge
- Mercury: Solver backends



Design Goals for Zinc

- Natural Modelling: clear and concise high-level mathematical models
- Extensible Modelling: support modelling for a wide-variety of applications by extending the modelling language
- Software Engineering: support the development of correct and maintainable models
- Practical Solver-Independence: allow a single conceptual model to be mapped to many design models

Designing Zinc

- Three years of work
- Many refinements and clarifications
- Many Zinc models written and reviewed
- Reconsidered decisions in light of experience
- Still refinements to be made!
- Highlighting some of the important decisions made/discovered

Outline

1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. Natural Modelling
4. Extensible Modelling
5. Software Engineering
6. Practical Solver-Independence
7. Conclusion

Modelling Combinatorial Optimization Problems

- Specification Languages
  - Generic specification languages: e.g. Z, B
    - More expressive than Zinc
    - Turing-complete, require theorem proving
    - Too powerful for combinatorial optimization
  - DIMACS SAT representation
    - Designed for solvers, not for modelling
    - Application and algorithm independent
    - Difficult to encode some problems: pigeonhole
  - MPS (lower level mathematical programming specification)
    - Again designed for solvers, not for modelling
    - Application and algorithm independent
    - Difficult to encode some problems: zebra
  - CSP $< V, D, C >$ variables V, domains D, constraints C
    - Expressive, algorithm independent
    - Limited modelling features (extensive constraint defn!)
  - No separation of data and model

Modelling Combinatorial Optimization Problems

- Modelling Languages
    - Mathematical modelling languages: MOLGEN, AMPL, GAMS
        - Support a range of mathematical programming solvers
        - Arrays of variables, iteration, separation of model and data
        - Restricted to linear arithmetic
    - Constraint logic programming languages: Eclipse, CHIP
        - Solver independence (Eclipse)
        - Full programming langugges
        - Untyped and procedural
    - Comet
        - Procedural Turing-complete language
        - Allows specification of conceptual and design models
        - Currently restricted to local search

Modelling Combinatorial Optimization Problems

- Modelling Languages
  - OPL
    - Inherits good features from math modelling langauges
    - Enumerated domains, type declarations, data structures, reification
    - Discrete and continuous variables
  - ESRA
    - Very high level: set and relation variables
    - Discrete variables only
  - Essence
    - Very high level: set, multiset, relation and function variables
    - Strongly typed
    - Discrete variables only

Modelling Combinatorial Optimization Problems

	CSP	AMPL	Comet	ECLiPSe	OPL	ESRA	ESSENCE	Zinc
Decidable	*Yes*	*Yes*	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Typed	-	-	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
High-level	-	-	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Con-types	-	-	-	-	-	-	-	*Yes*
Coercions	-	-	-	-	-	-	-	*Yes*
Extensible	-	-	*Yes*	*Yes*	-	-	-	*Yes*
Sep-model	-	*Yes*	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Platforms	-	*Yes*	-	*Yes*	-	-	*Yes*	*Yes*
Domains	E	CD	DBS	CDEBS	CDE	DEBS	DEBS	CDEBS

- Decidable: all models are solvable.
- Typed: language is typed.
- High-level: admits data structures such as records and sets.
- Con-types: constraints can be associated with (all variables/values of) a type.
- Coercions: support for both overloading and type coercions.
- Extensible: core language provides extensible features.
- Sep-model: model and data can be provided separately.
- Platforms: models can be mapped to different underlying platforms.
- Domains: supported constraint domains—continuous arithmetic (C), discrete arithmetic (D), discrete symbolic (E), Booleans (B), sets (S).

Outline

1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. **Natural Modelling**
4. Extensible Modelling
5. Software Engineering
6. Practical Solver-Independence
7. Conclusion

Example: Perfect Squares `perfsq.zinc`

```
type PosInt = (int: x where x > 0);
PosInt: base;
type Square = record(var 1..base: x, var 1..base: y, PosInt: size);
list of Square: squares;

predicate nonOverlap(Square: s, Square: t) =
              s.x + s.size <= t.x  \/  t.x + s.size <= s.x  \/
              s.y + s.size <= t.y  \/  t.y + s.size <= s.y;

constraint forall(s in squares) (
               s.x + s.size < base  /\  s.y + s.size < base  );
constraint forall(i, j in 1..length(squares) where i < j) (
               nonOverlap(squares[i], squares[j])         );

constraint assert(sum(s in squares)(s.size * s.size) == base*base,
               "Squares do not cover the base exactly");

solve satisfy;
output show(squares);
```

Example: Perfect Squares `perfsq.zinc`

- Constrained type item: `type PosInt = (int:  x where x > 0);`
- Parameter declaration item: `PosInt:  base;`
- Record type declaration item

  ```
  type Square = record(var 1..base:  x, var 1..base:  y,
                       PosInt:  size);
  ```
- (Array) Variable declaration item: `list of Square:  squares;`
- Predicate definition item

  ```
  predicate nonOverlap(Square: s, Square: t) =
                s.x + s.size <= t.x  \/  t.x + s.size <= s.x  \/
                s.y + s.size <= t.y  \/  t.y + s.size <= s.y;
  ```

Example: Perfect Squares `perfsq.zinc`

- Constraint items

```
constraint forall(s in squares) (
                s.x + s.size < base  /\  s.y + s.size < base  );
constraint forall(i, j in 1..length(squares) where i < j) (
                nonOverlap(squares[i], squares[j])         );
```

- Assertion

```
constraint
    assert(sum(s in squares)(s.size * s.size) == base*base,
           "Squares do not cover the base exactly");
```

- Solve item: `solve satisfy;`
- Output item: `output show(squares);`

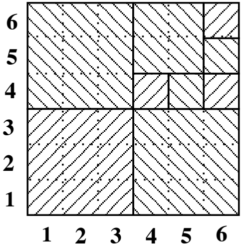
Example: Perfect Squares Data `perfsq6.data`

- Problem instance defined by separate data file
- Example

```
base = 6;
squares = [ (x:_, y:_, size:s) | s in [3,3,3,2,1,1,1,1,1] ];
```

- _ represents anonymous variable.
- Result

```
[(x:1, y:1, size:3), (x:4, y:1, size:3),
 (x:1, y:4, size:3), (x:4, y:5, size:2),
 (x:4, y:4, size:1), (x:5, y:4, size:1),
 (x:6, y:4, size:1), (x:6, y:5, size:1),
 (x:6, y:6, size:1)]
```



Types and Insts

- Types
  - Base: Booleans, integers, floats, strings
  - Constructors: sets, arrays, tuples, records, variant records, enumerated types
- Instantiations (Insts)
  - `par`: parameter — fixed by the data
  - `var`: decision variable
  - default is `par`
- Type-Inst (pairing of type and instantiation)

Design Decision: Type-Insts

- Modellers must distiguish between parameters and decision variables
- Allows: checking of parameter initialization, translation simplification
- Improves: readability, error checking

Base types

- Numbers
  - integers `23` and floats `2.3e-05`, `0.0067`
  - built-in arithmetic: `+`, `*`, `round`, ...
- Booleans
  - `true`, `false`
  - built-in operators: `\/`, `/\`, `->`, ...

Design Decision: Booleans

- Dont represent Booleans as 0..1 integers
- Better error checking, easier to map to different solvers

- Strings
  - `"one two three \n"`

Enumerated types and Sets

- Enumerated types
  - `enum Colour = {Red, Green, Blue};`
    `array[Colour,Colour] of var Colour:  Clashing;`

Design Decision: Enumerated types

- Name space for elements is global
- Type name is a set expression
- Can be declared in a model and defined in data file

  - `for(i,j in Colour)(Clashing[i,j] != i)`

Sets and Comprehensions

- Sets
  - sets literals `{1.0,-5.3}` and ranges `1..4`
  - built-in operators: `in`, `card`, `union`, `intersect`, ...
  - Only sets of `par` instantations
  - Var sets must have finite type
    - Yes: `var set of Colour`, `set of tuple(int,int)`
    - No: `set of var Colour`, `var set of tuple(int,int)`
  - set comprehensions
    - `{i * j | i,j in 1..10 where i != j }`

Design Decision: Comprehensions

- Generator sets of arrays must be `par`
- Ensures finiteness, some confusion

- No: `var set of 1..10:  p; constraint sum(i in p)(i) > 0;`
- Yes: `constraint sum(i in 1..10)(bool2int(i in p)*i) > 0;`

Arrays

- Arrays
  - Elements with any instantiation
  - Indices must be `par`, arrays are never `var`

Design Decision: Arrays

- Arrays of variable length are disallowed (finiteness)
- Multi-dimensional arrays are actually arrays indexed by tuple
- Lists are syntactic sugar for arrays

  - `for(i,j in Colour)(Clashing[(i,j)] != i)`
- Explicit and implicit indices
  - By default integral beginning at 0

  -
    ```
    array[1..3]   of int: a1 = [5, 6, 7];  % explicit-index
    array[int]    of int: a2 = [0:8, 1:9]; % implicit-index
    array[1..100] of int: a3 = [1:1, 2:2] default 0;
    ```
- array comprehensions
  - `[i * j | i,j in 1..10 where i != j ]`

Tuples, Records and Variant Records

- Tuples
  - Elements of any instantiation
  - `tuple(int,float,bool):  x = (1, 4.56, true);`
  - Field numbers: `x.3 \/ y >= 0`
- Records
  - Elements of any instantiation
  - `record(x:int, y:float, z:bool):  y = x;`
  - Field access: `y.1 >= 8`
  - Coercion from tuples of correct type.
- Variant Records
  - Non-recursive

  -
    ```
    variant_record thing = {
                        integer(int:x),
                        boolean(bool:b),
                        pair(int:x.  int:y) };
    ```

Let Expressions

- Let expressions
  - Introduce local variables and parameters
  - `let { int:x = z*z, var int:y = u*u } in x + y`

Design Decision: Let

- Let expressions in negative contexts must functionally define variables
- Otherwise: requires universal quantification

  - No: `not (let {var int:z } in x == 2 * z)`
  - Yes: `not (exists(z in 1..10)(x == 2 * z))`
  - Yes: `not (let {var int:z = floor(x / 2) } in x == 2 * z)` is equivalent to `let {var int:z = floor(x / 2) } in not(x == 2 * z)`

Items

- Assignment items
  - `x = 3;`
  - Joint variable declarations are sugar: e.g. `int:x = 3;` ⇒ `int:x; x = 3;`
- Include items
  - `include "globals.zinc"`
  - Textually insert file into incliding file

Design Decision: Items

- Items can appear in any order (helps include)
- Data files are just Zinc (complex expressions allowed)

-
```
enum Colour;
Colour: None;
int: x;
var Colour:y;
include "test.data"
constraint y != None;
```

```
enum Colour = { Empty, Red, Blue };
None = Empty;
x = card(Colour) - 1;
```

test.data

Outline

Extensible Modelling

- Rich type language
  - Define a new type for some application domain
- User-defined predicates and functions
  - Define the operations (functions) for the new type
  - Define the constraints (predicates) for the new type
- Constrained types
  - Enforce certain constraints on all members of the type

Predicates and Functions

- Predicates are simply functions with return type `var bool`
- Functions can be overloaded on type-inst
- Types can use type parameters `$T`

```
function var bool: between($T: x, $T: y, $T: z) =
    (x <= y /\ y <= z) \/ (z <= y /\ y <= x);
function par bool: between(par $T: x, par $T: y, par $T: z) =
    (x <= y /\ y <= z) \/ (z <= y /\ y <= x);
```

Second version gives more accurate type-inst on return `par bool`

Design InDecision: Overloading

- Currently body is duplicated for overloading
- Later perhaps we allow sharing of bodies

foldl and foldr

- Two higher-order built-in functions
- `foldl(fun, init, array)`
  - Apply binary function *fun* to each element in *array* starting from *init*

```
predicate        forall(array[int] of var bool:  xs) =
                        foldl('/\', true, xs);
function var int:   sum(array[int] of var int:   xs) =
                        foldl('+', 0, xs);
function var float: sum(array[int] of var float: xs) =
                        foldl('+', 0, xs);
```

Design Decision: Fold

- Provide powerful building blocks `foldl`, `foldr`
- Preferable to many built-in iteration constructs `forall`, `sum`

Reflection functions

- In order to define functions and predicates
- Reflection functions
  - `index_set`: returns index of array argument
  - e.g. `index_set_1of2`: first index of 2d array
  - `lb`: returns declared lower bound of `var int`, `var float`
  - `dom`: returns declared domain of `var int`, `var float`
  - `ub`: returns declared upper bound of `var int`, `var float`, `var set of ..`

```
%-----------------------------------------------------------%
% Requires the image of function 'x' (represented as array)
% on set of values 's' is 't'
%-----------------------------------------------------------%
predicate range(array[int] of var int:x, var set of int:s,
                var set of int:t) =
    forall(i in ub(t))(
        i in t -> exists(j in ub(s))(j in s -> x[j] == i));
```

Constrained Types

- All elements of the type must satisfy a certain constraint

```
type PosInt   = (int: i where i > 0);
type Interval = (record(var int: start, var int: end):
                          r where r.end >= r.start);
```

- ranges `1..n` are a special case
- Constraint view: syntactic sugar, e.g. `var PosInt:  y;` ⇒

```
var int:  y;
constraint y > 0;
```

- Type theoretic view: subtype not active constraint
  - should not affect execution of type correct program
- The constraint view and subtype view are incompatible!

Constrained Types Difficulties

- `predicate ge(var PosInt:x, var PosInt:y) =  x >= y;`

  ```
  var int: x = 5;  var int: y = -6;  var int: z;
  constraint ge(x,y) \/ z = 1;
  ```

- subtype view: type error!
- constraint view: `z = 1` since equivalent to

  ```
  constraint (x > 0 /\ y > 0 /\ x >= y) \/ z = 1;
  ```

- Problematic with negation!

  ```
  constraint ( ge(x,y) /\ z = 1) \/ (not ge(x,y) /\ z = 2);
  ```

  - Constraint view: $x = 5 \wedge y = -6 \wedge z = 2$
  - subtype view: type error

Design Decision: Constrained types

- Check statically whether formal types implied by actual types
- Warn if not so, and use constraint viewpoint

Outline

1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. Natural Modelling
4. Extensible Modelling
5. **Software Engineering**
6. Practical Solver-Independence
7. Conclusion

Software Engineering

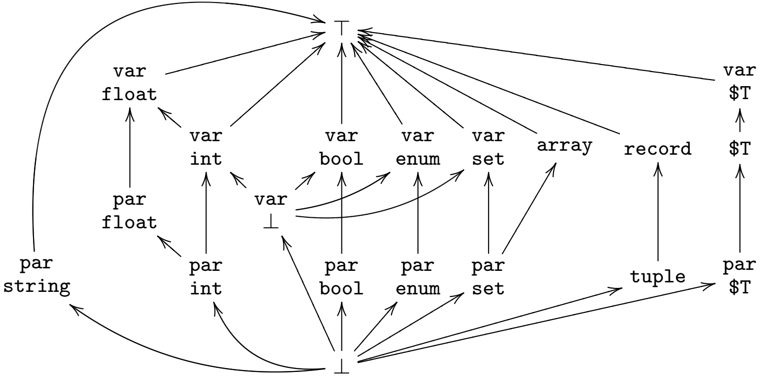
- Zinc designed to support good software engineering practices
  - Conciseness and readability
  - Strong static error checking
  - Avoiding code duplication (parametricity)
- Problem instance evaluation
  1. *model checking*: data-independent check of model
  2. *instance checking*: checking once the data is combined with the model
  3. *instance evaluaion*:
- Zinc checks errors as soon as possible in the pipeline

Design Decision: Variable Declarations

- All variables except generator variables must have a declared type-inst
- Complete inference impossible with separate data files

Type-Inst Checking

- Main static check
- Type system: Hindley-Milner with coercions and overloading
- Type-inst lattice



Type-Inst Checking

- Coercions
  - Integers can be coerced to floats
  - Sets S of type t can be coerced to arrays indexed by $0..card(s)$ of type t
  - Tuples can be coerced to records (if type appropriate)
  - `par` values can be coerced to `var` values
  - Anonymous variable `_` (type-inst `var` $\perp$) can be coerced to any `var` type-inst.
- Algorithm
  - Bottom-up inference: determine type-inst
  - Top-down: add appropriate coercions

Type-Inst Checking Example

- `array[int] of var float:  x = [1,2,_]`
- array element type: lub of `par int`, `var` $\perp$ = `var int`

```
[ coerce(par int,    var int, 1),
  coerce(par int,    var int, 2),
  coerce(var bottom, var int, _) ];
```

- Assignment means coerce `array[int] of var int` to `array[int] of var float`

```
array[int] of var float: x =
    coerce(array[int] of var int, array[int] of var float,
           [ coerce(par int,    var int, 1),
             coerce(par int,    var int, 2),
             coerce(var bottom, var int, _) ];
```

- Coercions are pushed down as far as possible

```
array[int] of var float: x =
           [ coerce(par int,    var float, 1),
             coerce(par int,    var float, 2),
             coerce(var bottom, var float, _) ];
```

Overloading

- Overloading interacts dangerously with coercion!

```
function int: f(int: x, float: y) = 0;
function int: f(float: x, int: y) = 1;
```

  Type of `f(3,3)` determines result 0 or 1!

- Type-inst checking with overloading and coercion may be expensive

```
function int:   g(int: x,   float: y);
function float: g(float: x, int: y);
```

  Checking `g(g(g(1,2),g(3,4)),g(5,g(6,7)))` is combinatorial

Design Decision: Overloading

- Overloaded versions of functions should be semantically equivalent wrt coercion: Not statically checkable
- Overloaded functions must be closed under type conjunction
- Overloaded functions must be monotonic

Array access, Domain checking, Assertions

- Array access (`var`ifying)
  - `array[int] of par int:  x = [1,2,3];`
  - `var int:i;` then `a[i]` has type-inst `var int`
- Domain checking
  - Set solvers require sets to range over finite domains
  - *finite type*: enumerated type, range, Boolean: tuples, records, sets of finite types.
- Assertions
  - Two assertion functions

    ```
    function $T: assert(par bool: c, par string: s, $T: val);
    function par bool: assert(par bool: c, par string: s);
    ```
  - Check `c`, if false print `s` else return `val` (or true)

Design Decision: Assertions

- Originally assertion item, expressions are more flexible, particularly for functions

Outline

1. Design Goals for Zinc
2. Modelling Combinatorial Optimization Problems
3. Natural Modelling
4. Extensible Modelling
5. Software Engineering
6. Practical Solver-Independence
7. Conclusion

Practical Solver-Independence

- Zinc is designed to allow the mapping of a conceptual model to different design models.
- Features to support this are:
    - Annotations
    - Decomposable Global Constraints
    - Implementability

Annotations

- Annotations can be used to control the translation of conceptual models
- Solver dependent information (this includes search!)
- Can be ignored by a solver
- Declared with types:

```
enum SolverKind = { Lp, Ip, Fd, Sat };
annotation solver(SolverKind);
annotation bounds;
```

- Attached to expressions and items with `::`

```
var array[int] of var int: x :: bounds;
constraint all_different(x) :: solver(Fd) :: bounds;
constraint sum(i in S)(a[i] * x[i]) <= 10 :: solver(Lp);
```

Decomposable Global Constraints

- Global constraints are simply predicates
- Low-level logical definition means they can be used for any solver!
- Example: sequence constraints: in each sequence of length `k` in `x` there are between `l` and `u`, 1s:

```
predicate sequence_among(int: l, int: u, int: k,
                         array[int] of var 0..1: x) =
    let { int: n = max(index_set(x)) } in
    assert(min(index_set(x)) == 1 /\ card(index_set(x)) == n,
        "array x must be indexed 1..n",
    forall(i in 1..n-k+1)(
        among(l, u, [x[j] | j in i..i+k-1])));

predicate among(int: l, int: u, array[int] of var 0..1: x) =
    let { var int: s = sum(x) } in l <= s /\ s <= u;
```

- Supports experimentation with different decompositions, e.g. `sequence_cumul`

Implementability

Zinc is designed so that the features are mappable to modern CP solvers

- Evaluate parameters
- Determine initial domain for all decision variables
- Simplify records to tuples, flatten tuples, replace field accesses
- Replace enumerated types by integer range types
- Unfold built-in and defined predicates and functions
- Insert constraints arising from constrained types
- Lift lets to be global (rename variables)
- Simplify arrays to be one dimensional integer indexed from 0
- Translate variables sets on structured types to `var set of int`
- Reify to separate logical combinations of constraints

Implementations

- Two implementations
- Prototype Full Zinc compiler
  - 12000 Mercury LOC, 5000 C LOC
  - Generates simplifed Zinc using translation on previous slide
  - Maps simplified Zinc to Eclipse: 3 solver
    - Complete tree search with FD propagation
    - (Repair-based) Local search maintaining some hard constraints
    - Mapping to MIP and branch and bound search
- In progress "Industrial Strength" Zinc compiler
  - 25000 Mercury LOC
  - Syntax and semantics checks
  - Transformation using Cadmium of a subset of Zinc, MiniZinc, to FlatZinc
  - FlatZinc interpretable by LP/FD solvers, and Gecode, Eclipse

Outline

Conclusion

Zinc

- Allows clear and concise high-level mathematical models
- Extensible with constrained types, and user-defined predicates and functions
- Supports the development of correct and maintainable models
- Allows a single conceptual model to be mapped to many design models

Future

- Many mapping of Zinc to solvers to explore
- Specifying search in Zinc
- Direct mapping of Zinc to Mercury

TRY OUT MiniZinc!

www.g12.mu.oz.au/minizinc/