# The Modelling Language Zinc

Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace

Clayton School of IT, Monash University, Australia
{mbanda, marriott, reza.rafeh, wallace}@mail.csse.monash.edu.au

**Abstract.** We describe the Zinc modelling language. Zinc provides set constraints, user defined types, constrained types, and polymorphic predicates and functions. The last allows Zinc to be readily extended to different application domains by user-defined libraries. Zinc is designed to support a modelling methodology in which the same conceptual model can be automatically mapped into different design models, thus allowing modellers to easily "plug and play" with different solving techniques and so choose the most appropriate for that problem.

## 1 Introduction

Solving combinatorial problems is a remarkably difficult task which requires the problem to be precisely formulated and efficiently solved. Even formulating the problem precisely is surprisingly difficult and typically requires many cycles of formulation and solving. Efficiently solving it often requires development of tailored algorithms which exploit the structure of the problem, and extensive experimentation to determine which technique or combination of techniques is most appropriate for a particular problem. Reflecting this discussion, modern approaches to solving combinatorial problems divide the task into two (hopefully simpler) steps. The first step is to develop the *conceptual model* of the problem which specifies the problem without consideration as to how to actually solve it. The second step is to *solve* the problem by mapping the conceptual model into an executable program called the *design model*. Ideally, the same conceptual model can be transformed into different design models, thus allowing modellers to easily "plug and play" with different solving techniques [4].

Here we introduce a new modelling language, Zinc, specifically designed to support this methodology. There has been a considerable body of research into problem modelling which has resulted in a progression of modelling languages including AMPL [2], Localizer [6], OPL [7], and specification languages including ESRA [1] and ESSENCE [3]. We gladly acknowledge the strong influence that OPL has had on our design. Our reasons to develop yet another modelling language are threefold.

First, we want the modelling language to be solver and technique independent, allowing the same conceptual model to be mapped to different solving techniques and solvers, i.e., be mapped to design models that use the most appropriate technique, be it local search, mathematical modelling (MIP), Constraint Programming (CP), or a combination of the above. To date the implemented

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 700–705, 2006.
© Springer-Verlag Berlin Heidelberg 2006

languages have been tied to specific underlying platforms or solving technologies. For example, AMPL is designed to interface to MIP packages such as Cplex and Xpress-MP, Localizer was designed to map down to a local search engine, and ESSENCE and ESRA are designed for CP techniques. Of the above, only OPL was designed to combine the strengths of both MIP and CP, and now the most recent version of OPL only supports MIP.

Second, we want to provide high-level modelling features but still ensure that the Zinc models can be refined into practical design models. Zinc offers structured types, sets, and user defined predicates and functions which allow a Zinc model to be encapsulated as a predicate. It also allows users to define "constrained objects" i.e., to associate constraints to a particular type thus specifying the common characteristics that a class of items are expected to have [5]. It supports polymorphism, overloading and type coercion which make the language comfortable and natural to use. However, sets must be finite, and recursion is restricted to iteration so as to ensure that execution of Zinc programs is guaranteed to terminate. Zinc is more programming language like than the specification based approaches of ESSENCE and ESRA. These provide a more abstract kind of modeling based on first-order relations. Currently, they do not support variables with continuous domains or, as far as we can tell, functions or predicates. Furthermore, only limited user-defined types are provided.

And third, we want Zinc to have a simple, concise core but allow it to be extended to different application areas. This is achieved by allowing Zinc users to define their own application specific library predicates, functions and types. This contrasts with, say, OPL which provides seemingly ad-hoc built-in types and predicates for resource allocation and cannot be extended to model new application areas without redefining OPL itself since it does not allow user-defined predicates and functions.

## 2 Zinc

Zinc is a first-order functional language with simple, declarative semantics. It provides: mathematical notation-like syntax; expressive constraints (finite domain and integer, set and linear arithmetic); separation of data from model; high-level data structures and data encapsulation including constrained types; user defined functions and constraints.

As an example of Zinc, consider the model in Figure 1 for the perfect squares problem [8]. This consists of a base square of size `sizeBase` (6 in the figure) and a list of squares of various sizes `squares` (three of size 3, one of size 2 and five of size 1 in the figure). The aim is to place all squares into the base without overlapping each other.

The model defines a constrained type `PosInt` as a positive integer and declares the parameter `sizeBase` to be of this type. A record type `Square` is used to model each of the squares. It has three fields `x`, `y` and `size` where $(x, y)$ is the (unknown) position of the lower left corner of the square and `size` is the size of its sides. The first constraint in the model ensures each square is inside the base (note

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

constraint  forall(s in squares)
              s.x + s.size =< sizeBase+1 /\
              s.y + s.size =< sizeBase+1;
predicate nonOverlap(Square: s,t) =
              s.x+s.size =< t.x  \/  t.x+t.size =< s.x \/
              s.y+s.size =< t.y  \/  t.y+t.size =< s.y;
constraint forall(i,j in 1..length(squares) where i<j)
              nonOverlap(squares[i], squares[j]);
predicate onRow(Square:s, int: r) =
              s.x =< r /\ r < s.x + s.size;
predicate onCol(Square:s, int: c) =
              s.y =< c /\ c < s.y + s.size;
assert sum(s in squares) (s.size^2) == sizeBase^2;
constraint  forall(p in 1..sizeBase)
              sum(s in Squares) (s.size*holds(onRow(s,p))) == sizeBase /\
              sum(s in Squares) (s.size*holds(onCol(s,p))) == sizeBase;

output(squares);
```

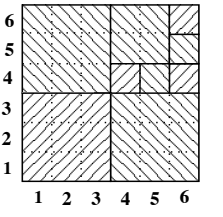


**Fig. 1.** Perfect Squares model

that \/ and /\ denote disjunction and conjunction, respectively). The model contains three user-defined predicates: `nonOverlap` which ensures two squares do not overlap, while `onRow` and `onCol` ensure the square is, respectively, on a particular row or column in the base.

The squares provided as input data are assumed to be such that they fit in the base exactly. To check this assumption, the model includes an assertion that equates their total areas.

The last constraint in the model is redundant since it is derived from the assumption that the squares exactly fill the base: the constraint simply enforces each row and column in the base to be completely full.

Data for the model can be given in a separate data file as, for example:

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

Let us now look at the more interesting features of Zinc.

**Types:** Zinc provides a rich variety of types: integers, floats, strings (for output), Booleans, arrays, sets, lists, tuples, records, discriminated union (i.e. variant records) and enumerated types. All types have a total order on their elements, thus facilitating the specification of symmetry breaking and of polymorphic predicates and functions. In the case of compound types this total order is the natural lexicographic ordering based on their component types.

A useful feature of Zinc is that arrays are not restricted to integer indexes, they are actually maps from virtually any type to any other type. Similarly, Sets can be of any data type as long as they are finite.

In order to allow natural mathematical-like notation, Zinc provides automatic coercion from integers to floats, from sets to lists, from lists to arrays and from tuples to records with the same field types. A set is coerced to a list with its elements in increasing order, and a list of length $n$ is coerced to an array with index set $1..n$.

One of the novel features of Zinc is that types, such as `PosInt` in the perfect squares model in Figure 1, can have an associated constraint on elements of that type. Effectively, whenever a variable is declared to be of constrained type, the constraint is implicitly placed in the model.

**Variables:** All variables must be declared with a type except for local variables occurring in an array, list or set comprehension. The reason for requiring explicit typing is that automatic coercion and separate datafiles precludes complete type inference.

Variables have an associated *instantiation* which indicates whether they are *parameters* of the model whose value is known before performing any solving of the model, or *decision variables* whose value is known only after. Variables are, by default, assumed to be parameters, and are declared to be decision variables by adding the `var` keyword before their type definition. This keyword can only be applied to integers, floats, enumerated types, Booleans, and sets. Lists of variable length are not allowed.

**Expressions:** Zinc provides all the standard mathematical functions and operators, many of which are overloaded to accept floats and integers.

List, set and array comprehensions provide the standard iteration constructs in Zinc. Examples of their use are shown in the preceding example. Other iterations such as `forall`, `exists`, `sum` and `product` are defined as Zinc library functions using $foldl(Fun, List, Init)$, which applies the binary function $Fun$ to each element in $List$ (working left-to-right) with initial accumulator value set to $Init$. For instance,

```
constraint  forall(list of bool: L) = foldl('/\',L,true);
function int: sum(list of int: L) = foldl('+',L,0);
function float: sum(list of float: L) = foldl('+',L,0);
```

$foldl$ and $foldr$ are the only higher-order functions provided by Zinc. User-defined higher-order functions and predicates are not allowed in Zinc.

**Constraints:** Zinc supports the usual constraints over integers, floats, Booleans, sets and user defined enumerated type constants. All constraints, including user-defined constraints, are regarded as Boolean functions and can be combined using the standard Boolean operators. Higher-order constraints can also be readily defined, which is useful, for example, to define functions such as the built-in function *holds* which returns 1 if the constraint holds, and 0 otherwise.

The standard global constraints, such as `alldifferent`, are provided. Note that thanks to the existence of a total order on the elements of any type, the `alldifferent` global constraint works for lists of any type, including records.

**User-defined Predicates and Functions:** One of the most powerful features of Zinc is the ability for users to define their own predicates and functions, such as `nonOverlap`, `onRow` and `onCol`, in the perfect square model in Figure 1. Zinc supports polymorphic types and context-free overloading. Although the average modeller may not use these facilities, it allows standard modelling functions to be defined in Zinc itself. We have previously seen an example of how the library function `sum` is overloaded to take either a list of integers or a list of floats, and how the library function `alldifferent` is polymorphically defined for lists of any type. As another example of polymorphism, consider the polymorphic predicate `between` (with polymorphic types being indicated by `$T`):

```
predicate between($T: x,y,z) =
       (x =< y /\ y =< z) \/ (z=<y /\ y=<x);
```

which applies to numeric and non-numeric types, lists, tuples, records and sets!

User-defined functions and predicates are instantiation-overloaded in the sense that a definition can take both parameters and decision variables.

## 3 Conclusion and Future Work

We have presented a new modelling language Zinc designed to allow natural, high-level specification of a conceptual model. Unlike most other modelling languages, Zinc provides set constraints, constrained types, user defined types, and polymorphic predicates and functions. The last allows Zinc to be readily extended to different application domains by user-defined libraries.

One of the main aims of developing Zinc is that a Zinc model can be mapped into design models that utilize different solving techniques such as local search or tree-search with propagation based solvers. Currently, we are implementing three mapping modules to map the Zinc models into design models in ECLiPSe for three different solving techniques: constraint programming, local search and mathematical methods.

## References

1. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.
2. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002.
3. A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The essence of ESSENCE: A constraint language for specifying combinatorial problems. In *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.

4. C. Gervet. *Large scale combinatorial optimization: A methodological viewpoint*, volume 57 of *Discrete Mathematics and Theoretical Computer Science*, pages 151–175. DIMACS, 2001.
5. B. Jayaraman and P. Tambay. Modeling engineering structures with constrained objects. In *PADL*, pages 28–46, 2002.
6. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. Principles and Practice of Constraint Programming - CP97*, pages 237–251, 1997.
7. P. Van Hentenryck, I. Lustig, L.A. Michel, and J.-F. Puget. *The OPL Optimization Programming Language*. MIT Press, 1999.
8. E. W. Weisstein. Perfect square dissection. From MathWorld –A Wolfram Web Resource, http://mathworld.wolfram.com/PerfectSquareDissection.html, 1999.