Towards the new modelling language Zinc

R. Rafeh, K. Marriott, M. Wallace and M. Garcia de la Banda
Clayton School of IT, Monash University, 6800, Australia
{rezar,marriott,wallace,mbanda}@csse.monash.edu.au

Abstract

Combinatorial optimization problems are usually tackled in two steps: modelling and solving. Three main approaches are used for solving: Mathematical Methods (MM), Constraint Programming (CP) and Local Search (LS). For modelling the main tools are constraint programming languages, constraint programming libraries and (mathematical) modelling languages. Modelling languages provide the best approach to modelling for non-programmers since they do not require sophisticated programming skills. However, current modelling languages are tied to their underlying solvers and cannot support all three solving techniques. This is unfortunate since it is often not clear which technique is most suitable. This paper presents the preliminary design of *Zinc*, a new solver independent modelling language which is intended to support all three solving techniques.

1 Introduction

Combinatorial optimization problems appear in many real life applications, such as planning, scheduling, timetabling, routing, placement, investment, DNA sequencing, configuration, design and insurance. However, this class of problems is often NP-hard and, as a result, no efficient and general algorithms are known to solve them. The is unfortunate since their search space (i.e. the number of possible choices) grows exponentially with the number of the variables.

The main steps to handle combinatorial optimization problems are modelling and solving [1, 2]. The solving step uses three major techniques: Mathematical Methods (MM), Constraint Programming (CP) and Local Search (LS). Mathematical techniques include Linear Programming (LP), Integer Programming (IP) and Mixed Integer Programming (MIP). Mathematical techniques are efficient but unsuitable for some types of combinatorial optimization problems. CP techniques try to reduce the search time by pruning the search space. They are more flexible than mathematical techniques, but might require a huge amount

of time for solving some real problems. Furthermore, they are not suitable for optimization problems. LS techniques are very good for solving some of the problems for which other techniques need a substantial amount of time, because they do not explore the search space completely. Thus, they are not guaranteed to find a solution and, even if they do, it is unclear how far that solution diverges from the optimal one.

Regarding the modelling step, the tools most relevant to our current research are: constraint programming languages, constraint programming libraries and (mathematical) modelling languages. Constraint programming languages such as CLP(R) [4], ECLiPSe [6] and COMET [5] are generic programming languages which support solving techniques. They allow users to state their search strategies and define their own application specific constraints. Unfortunately, constraint programming languages lack support for high-level modelling and thus require the user to posses sophisticated programming skills, which widens the distance between the actual problem and the model representation. Constraint programming libraries such as CPLEX [7] and Modeller++ [8] are libraries within some (often object oriented) programming languages, which have some facilities for modelling. Their advantages are that users can integrate them in larger applications and do not need to learn a new language. However, these modelling tools cannot support high-level modelling either. Moreover, these libraries impose some additional restrictions, which are inherited from the underlying languages. Modelling languages such as AMPL [9], GAMS [10] and OPL [11] support a very high-level formalism with a syntax close to mathematical notation. Such syntax makes these languages accessible to users who are not computer scientists. However, since mathematical modelling languages are not general purpose programming languages, their expressiveness is limited.

¿From the previous discussion we can see that modelling languages provide the best approach to modelling for non-programmers since, unlike CP toolkits and languages, they do not require sophisticated programming skills. This is important because most modellers are not computer scientists. However, current modelling languages cannot support all three solving techniques: MM, CP and LS. This is unfortunate since each solving technique has its own advantages and disadvantages and it is unclear for a given model or application which technique or combination of techniques yields the best result. Thus, one should be able to create a single model and then solve it using different techniques to determine the best one.

This paper presents the preliminary design of *Zinc*, a new modelling language under development at Monash University. The main feature of *Zinc* is its support for solver independent modelling and all three solving techniques. In addition, *Zinc* supports expressive and natural modelling which makes it suit-

able for many real life problems. The rest is organized as follows: section 2 sets out the design of *Zinc*, section 3 presents a *Zinc* model for the open stacks problem and section 4 concludes the paper.

2 The Design Goals of *Zinc*

Our goals in the design of *Zinc* are to provide the following features:

1. An enhanced formalism for modelling real life problems easily
    - High-level syntax: A syntax close to mathematical notation. This includes: providing standard data structures such as sets and arrays; iteration; aggregated operators to provide operations on all members of sets, arrays and lists such as *sum*; comprehension to collect objects in sets, lists and arrays by using a formula as their relationship; overloaded operators and automatic coercion.
    - Expressive constraints: Linear and non-linear constraints, global constraints (to impose some constraints on a group of objects) and set constraints.
    - Soft and hard constraints: Ability to state which constraints in a model can be violated (i.e. are soft constraints) and which constraints must necessarily be satisfied (i.e. are hard constraints).
    - Support for optimization problems: Ability to define objective function(s).
2. Attention to software engineering issues
    - Improved reusability and extensibility: Separating data from model, ability to define new *constrained objects* (i.e. defining new objects which contain variable attributes with some constraints on them), new (polymorphic) functions and constraints and consequently, defining new libraries.
    - Reliability: Using assertions to check data consistency especially for the case in which the data is separated from the model (because the declaration of parameters are available in the model while they might be initialized in a data file), in addition to strong syntax and semantic checking of the model.

Additionally, the entire design of *Zinc* has been influenced by the trade off between expressiveness and efficiency, two orthogonal but often competing features.

3 A Short Example

This section provides a brief overview of *Zinc* by means of an example: a Zinc model for the minimisation of maximum number of open stacks [12]. This problem has been the focus of the 2005 Constraint Modelling Challenge competition [13]. In this scheduling problem, a manufacturer must satisfy a number of orders from customers. Each order is for a number of different products, and only one product can be made at a time. Once the first product in a customer's order is started, a stack is opened for that customer. When all the products required by a customer have been made, that order is sent and the stack is closed. Because of limited space in the production area, the maximum number of simultaneously open stacks should be minimized.

```
[1]  data "input.dat";
[2]  enum Customers;
[3]  enum Products;
[4] array[Products] of set of Customers:Ordered;
[5]  type time=1..length(Products);
[6]  array[time] of var Products:Assign;
[7]  array[time] of var set of Customers:OpenStacks
[8]  constraint alldifferent(Assign);
[9] constraint
[10] forall(t in time) OpenStacks[t] ==
[11] unionall({Ordered(Assign[ti]) | ti in 1..t})
[12] intersect
[13] unionall({Ordered(Assign[ti]) | ti in t..length(Products)})
[14] minimize max([|OpenStacks[t]| | t in time);
```

Figure 1: A *Zinc* model for the open stacks problem

Figure 1 shows a possible *Zinc* model for this problem. Line 1 provides the name of the data file ("input.dat") containing the input data associated to a particular instance of the problem. *Customers* and *Products* are enumerated types (defined in lines 2 and 3, respectively) used to store the customers' and products' names. *Ordered* is an array *parameter* containing for each product p, the name of all customers who ordered p. Note that in *Zinc* all parameters and enumerated types must be initialized either by the model itself, or by a data file. Since no initial values for *Ordered*, *Customers* or *Products* have been provided by the model, these values must appear in "input.dat".

Line 5 defines a new type *time* as a range between 1 and the number of

Products, while line 6 defines *Assign* as an array of *variables* assigning a product to each time. Variables need no initialization and are distinguished from parameters by using the *var* keyword in their declaration. When executing a model, *Zinc* tries to find for each variable a value that satisfies all the constraints in the model. Once the value of the variables in *Assign* has been determined, the array provides a possible time sequence for manufacturing the products. Line 7 defines *OpenStacks* as an array of variables assigning a set of customers to each time. For each sequence given by *Assign*, *OpenStacks* will contain the open stacks for those customers who are receiving their orders at each time in that sequence.

Line 8 declares a *constraint*. In *Zinc*, a constraint is a Boolean expression which comes after the *constraint* keyword. *alldifferent* is a global constraint which takes a list or an array and ensures that all elements of its argument have different values. Since the number of *Assign* elements equals the cardinality of each element domain, this constraint means that *Assign* takes a permutation of products.

Lines 9 to 13 ensure that, at any time t, $OpenStacks[t]$ contains those customers whose stacks are still open. This constraint uses *unionall*, a library function which takes a list of sets and returns their union. Its definition in the *Zinc* standard library is as follows:

```
function union2(set of $T: S1, S2):set of $T=
    S1 union S2;
function unionall(list of set of $T : L): set of $T=
    foldr(L,union2(),{});
```

The definition is based on the *union*2 function which takes two polymorphic sets (an identifier preceding by $ indicates a polymorphic type) and returns their union. *unionall* is then defined as a function which takes a list of polymorphic sets and returns a set of the same polymorphic type. It uses the built-in function $foldr$ to apply *union*2 to each element of the list and the current value of the accumulator, starting from the right-most element and the empty set as accumulator.

In our model for the open stacks problem, the first *unionall* returns the set of customers who ordered at least one product manufactured before or at time t. The second *unionall* returns the set of customers who have ordered at least one product manufactured after or at time t. The intersection between these two sets gives us the set of customers whose orders (and thus stacks) are still open.

Finally, line 14 shows an objective function which minimizes the maximum number of open stacks. *max* is a function which takes a list or a set and returns

the maximum member of that list or set. In this model, its argument is a generic list whose elements are the cardinality of associated open stacks to each time.

4 Conclusion

This paper presented the design goals of *Zinc*, a new solver independent modelling language that supports all three solving techniques: MM, CP and LS. In addition to the traditional features that other modelling languages provide, *Zinc* supports set variables, soft constraints, constrained objects, and user defined functions and constraints.

References

[1] K. Marriott , P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[2] R. Bartack. Constraint programming- What is behind? Proceedings of the workshop on constraint programming in Decision and control, June 1999, Poland.

[3] The Mercury project home page. http://www.mercury.cs.mu.oz.au/.

[4] J. Jaffar et al. The CLP(R) Language and System. ACM Transactions on Programming Languages and Systems, 14(3):339-395, July 1992.

[5] L. Michel, P. V. Hentenryck, L. Liu. Constraint-Based Combinators for Local Search. Proceedings of the 10th International Conference on Constraint Programming (CP-2003) Toronto, Canada, September 2004.

[6] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, M. g. Wallace. ECLiPSe An Introduction. Imperial College London 2003.

[7] CPLEX. Using the CPLEX callable library, Ver3. CPLEX Optimization Inc, Suite 279, Tahoe Blvd, Bldg 802, Incline Village, NV 89451-9436, 1995.

[8] L. Michel, P. V. Hentenryck. Modeler++: A Modelling Layer for Constraint Programming Libraries. CP-AI-OR'2001, Wye College (Imperial College), Ashford, Kent UK.

[9] R. Fourer, D. M. Gay ,B. W. Kernighan. AMPL: A Modelling Language for Mathematical Programming Management Science 36(1990) 519-554.

[10] R. E. Rosenthal. A GAMS Tutorial. Available from http://www.GAMS.com/docs/gams/tutorial.PDF.

[11] P. V. Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] A. Fink and S. Voss, Applications of modern heuristic search methods to pattern sequencing problems. Computers and Operations Research (26), pp. 17-34, 1999.

[13] B. M. Smith and I. P. Gent, Constraint Modelling Challenge 2005. Report on the entries to the 1st Constraint Modelling Challenge, 2005. http://www.dcs.st-and.ac.uk/ ipg/challenge/.