Journal of Computer Science & Systems Biology - Open Access

Research Article
OPEN ACCESS Freely available online
doi:10.4172/jcsb.1000063

JCSB/Vol.3 Issue 4

Solver Independent Modelling of Combinatorial and Optimization Problems

Reza Rafeh

Department of Computer Engineering, Islamic Azad University, Malayer Branch, Malayer, Iran

Abstract

Combinatorial optimization problems appear in many real life applications as timetabling, planning and scheduling. However, they are often NP-hard. This means that there is no general and efficient algorithm for solving them. Modern approaches for tackling combinatorial and optimization problems divide the task into two major tasks: modeling and solving.

Modelling means finding a proper formulation of the problem while solving means finding the solution of the problem. The most well-known modeling tools are: constraint programming languages, constraint libraries, (mathematical) modelling languages and specification languages. Modelling languages provide the most high-level practical level of modelling for modellers.

There are some known solving techniques to tackle such problems of which the most popular ones are: mathematical methods, constraint programming and local search. Each technique has its own advantages and disadvantages and for a given problem it is unclear at the beginning which technique gives us the best result.

Current modeling languages are tied to a specific solving technique. In this paper, we show how the modeling language Zinc can automatically map a conceptual model into corresponding low level model suitable for one of the aforementioned solving techniques.

Keywords: Combinatorial optimization problems; Zinc modelling language; Solver-Independence

There are many application areas in which combinatorial optimization problems appear: routing, placement, investment and DNA sequencing, to give a few examples. The main issue with this class of problems is the growth of their search space (i.e. the number of possible choices) exponentially with the number of the variables.

There are two major approaches to tackle combinatorial problems: developing a conceptual model of the problem which specifies the problem without consideration as to how to actually solve it and solving the problem by mapping the conceptual model into an executable program called the design model.

There are three main techniques for the solving step: Mathematical Methods (MM), Constraint Programming (CP) and Local Search (LS). Mathematical techniques (including Linear Programming (LP), Integer Programming (IP) and Mixed Integer Programming (MIP)) are efficient, but, finding a linear formulation of a problem is sometimes difficult. The aim of CP techniques is reducing the search time by pruning the search space. These techniques are more flexible than mathematical techniques, but might require a considerable amount of time for solving some real problems. Furthermore, they are not suitable for optimization problems. LS techniques avoid exploring the search space completely in the hope of reducing the solving time. On the negative side, there is no guarantee to find a solution. Even if they find a solution it is unclear how worse that solution is than the optimal one.

The most popular tools for modeling are: constraint programming languages, constraint programming libraries and (mathematical) modelling languages.

Constraint programming languages such as ECLiPSe (Apt and Wallace, 2007) are generic programming languages which support solving techniques. They allow users to state their search strategies and define their own application specific constraints. Unfortunately, using constraint programming languages is difficult for non-programmers.

Constraint programming libraries, such as Localizer++ (Michel and Hentenryck, 2001), 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 modeling 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 (Van Hentenryck et al., 1999), OPL (Fourer et al., 2002) and Localizer (Michel and Van Hentenryck, 1997) 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.

Specification languages such as ESSENCE (Frisch et al., 2007), aim at focusing on the problem without considering implementation details. While such languages provide the highest level of modelling, generic specification languages are usually impractical for constraint

Corresponding author: Reza Rafeh, Department of Computer Engineering, Islamic Azad University, Malayer Branch, Malayer, Iran, E-mail: reza_rafeh@yahoo.com

Received October 24, 2010; **Accepted** November 09, 2010; **Published** November 11, 2010

Citation: Rafeh R (2010) Solver Independent Modelling of Combinatorial and Optimization Problems. J Comput Sci Syst Biol 3: 086-088. doi:10.4172/jcsb.1000063

solving because of the large gap between their conceptual models and equivalent design models.

From the above discussion, it is clear that the most high-level practical modelling is provided by modelling languages. This is because, on the one hand, modelling languages do not require modellers to be skilled programmers (opposed to constraint programming languages and libraries) and on the other hand, they are implementable (as opposed to generic specification languages). However, existing 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. Therefore, modelers wish their models to be automatically mapped into different design models, thus allowing them to try different solving techniques for solving their models (Frisch et al., 2005).

Zinc (Marriott et al., 2008), is the first modeling language which in the one hand provides high-level formalism for modelers and on the other hand maps automatically a conceptual model to design models suitable for each solving technique. Zinc supports any (new) solver if the solver supports the required interface. In this paper, we explain how solver-independence has been achieved in Zinc.

Mapping Conceptual Models into Design Models

The huge gap between high-level Zinc models and low-level conceptual models is the main issue in achieving the aim of solver-independence in Zinc. To bridge this gap, we decided to use an intermediate modelling language, called Flattened Zinc. Flattened Zinc is a subset of Zinc designed to be simple and low-level enough to be significantly close to the design models, yet sufficiently high-level to specify suitable intermediate models for all solvers. It allows only simple constraints and data types. The advantage of first producing a flattened Zinc model (FZM) is that it allows us to perform mapping tasks that are common to all solving techniques, thus reducing the burden when developing mappings to new solvers and techniques.

The translation process from a conceptual model to different design models proceeds as follows. The first step takes a Zinc model and performs syntax and semantic analysis including type and model checking which includes inserting the required explicit coercions. The second step adds to the compiled Zinc model the information contained in the associated data file(s) (if any) and performs syntax and semantic checking (instance checking). The third step takes the model and data file(s) and generates the Solver-Independent Flattened Zinc Model (SI-FZM) instance. The fourth step uses solver-dependent rewrite rules to translate the SI-FZM into a Solver-Dependent Flattened Zinc Model (SD-FZM). As the name suggests, the rewrite rules used in this process depend on the target design model and the rewriting produces a Flattened Zinc model which is very close to the final design model. The final step takes the SD-FZM model and performs the minor syntactic rewriting required to generate the design model for a particular solving platform.

Our first implementation allowed a Zinc model to be mapped into one of the three design models, all of which are implemented in ECLiPSe. The first design model uses the standard constraint programming approach of a complete tree search with propagation based finite domain and set solvers. The second model is also complete but uses mathematical techniques, i.e. a MIP solver, while the third design model performs an incomplete search using local search methods. Importantly, users can execute the same Zinc model with each of these different solving techniques and so readily determine which technique is most appropriate for their particular problem.

Mapping to Flattened Zinc

As it was discussed previously, when flattening a Zinc model, we remove all high-level features which make Zinc user-friendly. More precisely, the only features of Zinc that are supported in the Flattened version are as the following (see (Rafeh et al., 2007) for more details):

- Types: Boolean, string, int, oat, set (only integer sets), list and tuple. Each type can be used with type-inst par or var (if applicable). Lists and tuples can be defined over other types.
- Expressions: Standard mathematical operations, function calls and standard Boolean operations.
- Items: Simple (reified) constraints, variable declarations, solve and output items.

As a result, any array, record, variant record, set over arbitrary types, enumerated and constrained type appearing in the Zinc model will be removed during the mapping to Flattened Zinc. Also, any negations, comprehension, function, predicate and propositional combination of constraints is removed during the mapping to Flattened Zinc.

Let us clarify this by means of an example. Figure 1 shows a Zinc model for the N-queens problem, which tries to place n queens on an n × n chess board in such a way that no two queens can attack each other.

Variable n is defined as an integer parameter. Domain is a new type for the range 1..n and q is an array of n finite domain decision variables (indicated by the keyword var) over that range. For our

```
int: n;
type Domain = 1..n;
array[Domain] of var Domain :q;
predicate noattack(Domain: i,j, var Domain: qi,qj) =
qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
constraint forall(i,j in Domain where i<j)
noattack(i,j,q[i],q[j]);
solve satisfy;
output(q);
```

Figure 1: A Zinc model for the N-queens problem.

```
var 1..4 : Q1, Q2, Q3, Q4 ;
constraint Q1 != Q2 ;
constraint Q1 + 1 != Q2 + 2;
constraint Q1 - 1 != Q2 - 2;
constraint Q1 != Q3 ;
constraint Q1 + 1 != Q3 + 3;
constraint Q1 - 1 != Q3 - 3;
constraint Q1 != Q4 ;
constraint Q1 + 1 != Q4 + 4;
constraint Q1 - 1 != Q4 - 4;
constraint Q2 != Q3 ;
constraint Q2 + 2 != Q3 + 3;
constraint Q2 - 2 != Q3 - 3;
constraint Q2 != Q4 ;
constraint Q2 + 2 != Q4 + 4;
constraint Q2 - 2 != Q4 - 4;
constraint Q3 != Q4 ;
constraint Q3 + 3 != Q4 + 4;
constraint Q3 - 3 != Q4 - 4;
solve satisfy;
output ([Q1,Q2,Q3,Q4]);
```

Figure 2: The FZM model for the N-queens problem (n=4).

Research Article
OPEN ACCESS Freely available online
doi:10.4172/jcsb.1000063
JCSB/Vol.3 Issue 4

purposes, the most interesting feature of Zinc is that it allows the user to define new predicates and functions. In the above example, the noattack predicate is defined, which succeeds if queens qi and qj of rows i and j respectively, cannot take each other (^ is used for conjunction). By using the forall expression, the constraint applies the noattack predicate on each pair of queens to ensue they cannot attack each other. The last line declares the model to be a satisfaction problem. If the solve item has no annotation for search (like in this model), Zinc uses the default search to solve the model. Finally, the output items prints the elements of array q.

This model includes some high-level features as arrays and predicates which must be simplified in the final FZM. Figure 2 depicts the equivalent FZM of the model. As it can be seen from the model, parameter **n** has been replaced with its value (4). Array q has been replace with 4 variables (Q1, Q2,Q3, Q4). Also, calls to predicates forall and notattack have been replaced with their bodies.

Conclusion

In this paper, we briefly discussed the idea behind developing the modeling language Zinc whose main aim was achieving solver-independence. In its first implementation, Zinc maps a conceptual model into design models that utilize different solving techniques such as local search, tree-search with propagation based solvers, or MIP techniques. The most crucial decision in implementing Zinc was using an intermediate language called Flattened Zinc which made the mapping simpler.

For the first implementation, we map Zinc models into low-level design models in ECLiPSe (Apt and Wallace 2006). One reason to choose ECLiPSe was its support of all solving techniques. To evaluate our mapping, we have compared a number of standard benchmarks written in Zinc and written in ECLiPSe. The ECLiPSe model automatically generated from Zinc (via FZM) has similar performance to an equivalent program written in ECLiPSe, using the same search method. This allows Zinc modellers to readily experiment with different solving techniques.

Acknowledgements

We thank Islamic Azad University, Malayer Branch for financial support of this project.

References

1. Apt K, Wallace M (2006) Constraint Logic Programming Using ECLiPSe. Cambridge University Press.
2. Apt K, Wallace M (2007) Constraint Logic programming Using ECLiPSe. Cambridge University Press.
3. Fourer R, Gay DM, Kernighan BW (2002) AMPL: A Modeling Language for Mathematical Programming. Duxbury Press.
4. Frisch AM, Jefferson C, Martinez-Hernandez B, Miguel I (2005) The rules of constraint modelling. In Proc 19th IJCAI 109-116.
5. Frisch AM, Grum M, Jefferson C, Martinez Hernandez B, Miguel I (2007) The design of ESSENCE: A constraint language for specifying combinatorial problems. In Proc. of the 20th International Joint Conference on Artificial Intelligence IJCAI.
6. Marriott K, Nethercote N, Rafeh R, Stuckey PJ, de la Banda MG (2008) The design of the Zinc modelling language. Constraints 13.
7. Michel L, Hentenryck PV (2001) Localizer++: An open library for local search. In Proc CS: 01-02.
8. Michel L, Van Hentenryck P (1997) Localizer: A modeling language for local search. In Proc Principles and Practice of Constraint Programming CP97: 237-251.
9. Rafeh R,Garcia de la Banda M,Marriott K,Wallace M (2007) From Zinc to Design Model. pages 215229. Number 4354 in LNCS. Springer, 2007. Proc. PADL 2007.
10. Van Hentenryck P, Lustig I, Michel LA, Puget JF (1999) The OPL Optimization Program-ming Language. MIT Press.