Journal of Advances in Computer Research
Quarterly pISSN: 2345-606x eISSN: 2345-6078
Sari Branch, Islamic Azad University, Sari, I.R.Iran
(Vol. 5, No. 4, November 2014), Pages: 1-8
www.jacr.iausari.ac.ir

On the Linearization of Zinc Models

Negar Jaberi[1], Reza Rafeh[2] *

1) Department of Computer Engineering Khomeinishahr Branch, Islamic Azad University Khomeinishahr/Isfahan, Iran
2) Department of Computer Engineering, Arak University Arak, Iran

negar.jaberi@iaukhsh.ac.ir; r-rafeh@araku.ac.ir

Received: 2013/7/27; Accepted: 2014/03/04

Abstract

Zinc is the first modelling language which supports solver and technique independence. This means that a high level conceptual model can be automatically mapped into an appropriate low level design model for a specific solver or technique. To date, Zinc uses three different techniques to solve a model: Constraint Programming (CP), Local Search (LS), and Mixed Integer Programming (MIP). In this way, modellers can examine all solving techniques for their models and see which one gives them the best result. MIP solvers can only accept linear models. Therefore, to map a conceptual model to MIP solvers, the model must be linearized first. In this paper we explain the techniques used in Zinc to linearize high level data structures and expressions which may be appeared in a conceptual model. As a result, modellers can benefit of expressive modelling using nonlinear expressions as well as efficiency of MIP solvers.

Keywords: *Linearization techniques, Linear programming, Zinc, Solver independence*

1. Introduction

Constraint decision problems appear in many applications such as scheduling, planning, routing [1] Solving such problems, which may be decreasing the cost or increasing the benefit of a company, plays a very important role in industry [2].

However, solving these problems is difficult because their search space grows exponentially in terms of the number of variables. As a result, using ordinary search algorithms may be inefficient for solving them[1].

Constraint decision problems must be modelled first which means an appropriate formulation of the problem must be found (conceptual model). A conceptual model usually includes variables, constraints and an objective function. Then, the given model must be solved which means that each variable gets an appropriate value such that all constraints are being satisfied and the objective function is being optimized [3].Well known techniques for solving constraint decision problems are Constraint Programming (CP), Mixed Integer programming(MIP), and Local Search(LS) [3].

MIP techniques can be applied only on linear models. Modelling with linear equations is more limited than with nonlinear ones. However, there are many linear solvers that can find an optimal solution for the given model efficiently. Therefore, mapping high-level nonlinear models into equivalent linear models allows the modellers

to enjoy both the precise solution of linear solvers and the high formalism of nonlinear models[4].

Zinc is a solver and technique independent modeling language which allows users to solve their models by three aforementioned solving techniques and see which one works better for their models. To do so, Zinc maps a conceptual model into an equivalent design model acceptable for the desired technique [5].

To solve a Zinc model by MIP techniques, the models must be linearized and high-level structures must be eliminated from the model. In this paper we explain the linearization techniques used by Zinc to perform this mapping.

The rest of the paper is organized as follows. Section 2 gives a Zinc model for n-queen problem to make the reader familiar with Zinc language. Section 3gives a brief overview of nonlinear expressions in Zinc. In section 4 we explain the techniques being used to linearize Zinc expressions and structures. In section 5 we evaluate our approach on a set of benchmarks. Finally, section 6 concludes the paper.

2. An Example of a Zinc Model

As an example, Figure 1 depicts a Zinc model for the n-queen problem. N queens must be placed in a n×n chess board such that no queen can take others. The first line defines a parameter n which is the number of queens. In line 2, an integer range as a new type is declared. In line 3 an array of variables is declared in which a variable shows the column of a queen. Constraints of the problem have been defined in lines 4-6, which for each two queens ensure they are neither in a same column nor in a same diagonal. The *all different* constraint is a global constraint which takes a list of variables and ensures that they have different values. Line 7 introduces the model as a satisfaction mode to Zinc. This means that the problem has no objective function [6].

```
1.  int: n;
2.  type Domain = 1..n;
3.  array[Domain] of var Domain :q;
4.  constraint alldifferent([Q[i]| i in 1..n]);
5.  constraint alldifferent([Q[i]+i| i in 1..n]);
6.  constraint alldifferent([Q[i]-i| i in 1..n]);
7.  solve satisfy;
```

Figure 1. *n-queen model in Zinc*

3. Nonlinear Expressions in Zinc

The most important feature of Zinc is that it can solve the model by all three methods, because it is not clear which technique provides the best solution. As a result, the users can choose their own solving method for their models .In order to be able to solve a model with the desired solving method, Zinc should convert the model into an acceptable form for the solver. One solving method is MIP which expects the model to be linearized first[5].

A MIP model is composed of variables, input parameters, the objective function and constraints.The mathematical definition of a MIP model implies the following assumptions:

1. *Divisibility assumption* for each continuous variable.
2. *Integrality assumption* for each integer variable.
3. *Certainty (constant) assumption* for each input parameter.
4. *Proportionality assumption* for each term in constraints and in the objective function.
5. *Additivity and separability assumption* for each combined function in the objective and constraints.
6. *Single-objective assumption*.
7. *Simultaneousness (conjunction)assumption* for all constraints [7].

However,the following Zinc expressions and operations do not confirm to MIP assumptions:

- Boolean operations
- Comparison operations
- Polynomials of degree two or above
- Arithmetic functions

Therefore, to use MIP solvers, Zinc must use linearization methods to transform existing nonlinear structures in a model into linear structures. Then, modellers can use advantages of linear solvers for solving their high-level models. This is important because for some problems, the solution obtained by MIP solvers may be better than other solvers [5].

4. Mapping to MIP

To map a conceptual Zinc model into a design model suitable for MIP solvers the original model must be linearized first. In the following sections we explain how such expressions are being linearized in a model intended to be solved by a linear solver.

4.1 Mapping Boolean operations to Mathematic Operations

A MIP model can consist only integer or real variables. Therefore all Boolean variables in a model are mapped to Binary variables [7]. Accordingly, the Boolean operations are mapped to equivalent operations on integer variables. For example, the following Zinc code:

```
var        bool:B1, B2;
constraint      B1 ∨ B2;
```
(1)

is transformed to the following Zinc code which is suitable for MIP [6]:

```
var        0..1:B1, B2;
constraint      B1 + B2 >= 1;
```
(2)

4.2 Mapping Comparison Operations to Mathematical Equations

Linear tools do not support dis-equality and strict inequalities: >, <, !=[1]. Therefore, they are transformed to the linear equations by using the reify function [5]. We use the

1. The dis-equality operator may be shown in other forms such as <> , \=.

Big M technique to translate comparison operators to equivalent expressions [8]. For instance, reify($X \leq Y$,B)[1] is modelled by the inequalities as follows [8]:

$$\begin{aligned}(X + B\times M \quad &\leq M + Y),\\ (X + M \quad &> (1 - B)\times M + Y)\end{aligned} \tag{3}$$

As a result, we can readily transform *reify(X < Y, B) to reify(X ≤ Y-e , B)* for an arbitrary small value ε and use the above equations to handle it.

Having all the above reify functions, we can implement *reify*(*X ¹ Y, B*) as follows[6] :

$$\begin{aligned}&reify\,(X < Y, B1)\\ &reify\,(X < Y, B2)\\ &B1 + B2 >= 1\end{aligned} \tag{4}$$

4.3 Mapping alldifferent Constraint

Global constraints are defined in Zinc as user-defined predicates in the Zinc standard library. If a solver directly supports a global constraint, the global constraint is mapped to the design model as it is, otherwise, it is replaced with the predicate body defined by the user. One useful global constraint is all different(L) which ensures all elements of list L have different values. An example of its usage was shown in the n-queen model in Section 2.

For linear version of the all different constraint, we assume that there are n elements in L and each element of L gets its value from domain [min,max]. We construct a binary matrix A with n rows and max-min +1 columns. Then, the all different constraint is mapped to the following constraints[6]:

$$\begin{aligned}&\forall i \in 1..n \sum_{j=\min}^{\max} a_{i,j} = 1 \forall i \in 1..n \sum_{j=\min}^{\max} a_{i,j} * j = L[i]\\ &\forall j \in min..max \sum_{i=1}^{n} a_{i,j} \leq 1\end{aligned} \tag{5}$$

4.4 Linearizing Multiplication of Binary and Real Variables

To linearize expression X*Y, where X is a binary variable and Y is a real variable, we introduce a new variable Z with which X*Y is replaced as follows [7]. Note that min and max are the lower and upper bound of Y.

$$\begin{aligned}&Z =< (X - 1) * \min + Y\\ &Z >= Y - (1 - X) * \max\\ &Z =< \max * X\\ &Z >= X * \min\end{aligned} \tag{6}$$

1. reify(C,B) ensures the binary variable B equals 1iff constraint C holds, otherwise B=0.

When X=1 the first two constraints ensure that Z=Y while the last two constraintsare released. When X=0, the last two constraints ensure Z=0, while the first two ones are released.

4.5 Linearizing Multiplication of Integer and Real Variables

To linearize a nonlinear expression X*Y in which X is an integer variable and Y is a real variable, a new variable Z is introduced to replace X*Y as follows. To do this, the integer variable X is mapped to an array of binary variables A. Then, using the technique mentioned for multiplication of binary and float variables mentioned previously, we place the multiplication of Y and every element of A in a float array T. Required equations are as follows .Note that min and max are the lower and upper bounds of X, respectively [5].

$W= \log_2 (max - min+1)$

$$\begin{aligned} &X = \min + \sum_{i=1}^{w} A_i \times 2^{i-1} \\ &\forall i \in 1..w \; T_i = A_i \times Y_i \\ &Z = \min \times Y + \sum_{i=1}^{w} T_i \times 2^{i-1} \end{aligned} \tag{7}$$

4.6 Linearizing Powers of Integer Variables

Since X^n equals to multiplication of X to itself n times, we can use the aforementioned techniques for multiplication to linearize it. However, it seems to be more efficient if we map X to a binary array having a value 1 in a position equals to X, and use the power operator to obtain X^n from the binary array as follows[9]:

$$\begin{aligned} &X = \sum_{i=\min}^{\max} A_i \times i \\ &Z = \sum_{i=\min}^{\max} A_i \times i^n \end{aligned} \tag{8}$$

4.7 Linearizing Products of Discrete and Integer Variables

A discrete variable can take only one value in a given list and does not confirm to the MIP format [7]. If X is a bounded integer variable and Y is a discrete variable which takes a value from a list L, to linearize X*Y,we use the following equations. As usual we assume min and max are the lower and upper bounds of X, respectively. Wb is a binary array in which there is only one bit equals to one whose position equals to the position of that element of L which equals to Y.

$W= \log_2 (max - min+1)$

$$
\begin{aligned}
1 &= \sum_{i=1}^{N} Wb_i \\
Y &= \sum_{i=1}^{N} Wb_i \times L_i \\
X &= \sum_{i=1}^{W} A_i \times 2^{i-1} \\
Z &= \sum_{i=1}^{N} \sum_{i=1}^{w} Wb_i \times L_i \times A_j \times 2^{j-1} + \sum_{i=1}^{N} Wb_i \times L_i \times \min
\end{aligned}
\tag{9}
$$

All we need is the multiplication of the value of Y (i.e., L_i if we assume Wb_i=1) and the value of X (i.e. $(\sum_{j=1}^{W} A_j \times 2^{j-1}) + min$). Adding min to the sigma is necessary because we map X to a binary representation with minimum value 0 [5].

5. Evaluation

The following problems have been used as benchmarks to evaluate the efficiency of the proposed approach:

- Design Template Problem [10] (2, 3 templates, 7 variations, 9 slots). The model has product of integer and real variables.
- Volume of Cylinder (with height in {50,100,200,250,300,350,400,450,500,550,600} and environment of circle in 2..200) [5]. This model includes both the second power and the product of discrete and integer variables.
- N-queen (18 queens). – This model contains alldifferent constraint.

The models have been executed on an Intel Core2 Duo CPU 2.20 GHz, 3GB of RAM running Microsoft Windows 8. To compare the execution time of the original (nonlinear) model with generated (linear) model, nonlinear models have been solved by propagation-based solvers using the first-fail technique [8] , while for linear models we used linear solvers using well-known linear algorithms including Simplex, Dual Simplex and barrier [11].

For Design Template problem, the linear model is faster when the number of templates is 2. However, when running the model for 3 templates, the execution time of the nonlinear model is better. Generally, although for the linear model the number of variables, about 9 times, and the number of constraints, about 10 times are greater than the nonlinear model, it seems for this problem the linearization process is worthwhile.

The linearized version of the Volume of Cylinder problem finds the optimal solution in a short time. Although the nonlinear model runs faster, it finds a sub-optimal solution.

The nonlinear version of n-queen is much faster than the linear one. In addition, the number of constraints and variables in the nonlinear model is less than the number of variables and constraints in the linear model.

Table 1. *Experimental results*

Problem	Type of model	No. Variables	No. Constraints	Execution time (s)
Design Template T = 2	Nonlinear model	16	22	**52.67**
	Linear model	142	242	**3.38**
Design Template T = 3	Nonlinear model	26	25	**667.93**
	Linear model	212	328	**782.84**
Volume of cylinder	Nonlinear model	2	2	**0.00**
	Linear model	376	535	**5.57**
Queens (18 queens)	Nonlinear model	18	3	**3.48**
	Linear model	1584	196	**1372.47**

6. Conclusions and Future Work

The Zinc modelling language supports solver independence which means a high-level model can be automatically mapped to different design models using different solving techniques. As a result, modellers can readily examine all solving techniques for their models to see which one gives the best result.

To solve Zinc models, which may include high-level structures, by MIP solvers, the original models must be mappedinto simpler equivalent models. In this paper we explained how Zinc provides necessary mappings to linearize nonlinear structures and expressions.Our experimental results showed that for some problems linearized models work better than the original nonlinear ones.

In the future, we intend to provide all well-known linearization techniques for Zinc in terms of Zinc functions and predicates instead of providing them as built-in in the compiler. In this way, the modeller can experience different linearization techniques for a given model and see which one works better for the model.

7. References

[1] K. Marriot, and P.J. Stuckey, *Programming with Constraint Programming: An Introduction*, The MIT Press, 1998.

[2] G. Ottosson, "Integration of Constraint Programming and Integer Programming for Combinatorial Optimization," Computing Science, Uppsala, 2000.

[3] M.G.d.l. Banda, K. Marriott, R. Rafeh, and M. Wallace, "The Modelling Language Zinc," *Principles and Practice of Constraint Programming*, Springer, pp. 700-705.

[4] N. Jaberi, and R. Rafeh, "A Survey of Linearization Techniques for Nonlinear Models," *International Journal of Computational Intelligence and Information Security*, vol. 3, no. 2, 2012, pp. 66-75.

[5] N. Jaberi, "Proposing Linearization Methods for Nonlinear Zinc Models," Computer Engineering, Islamic Azad University Malayer Branch, 2012.

[6] R. Rafeh, "The Modelling Language Zinc," Clayton School of IT, Monash, 2008.

[7] D.s. Chen, R.G. Batson, and Y. Dang, *Applied Integer Programming*, WILEY, 2010.

[8] R. Rafeh, M.G.d.l. Banda, K. Marriott, and M. Wallace, "From Zinc to design model " *Practical Aspects of Declarative Languages*, Springer, pp. 215-229.

[9] J.B.d. Fonseca, "Solving Any Nonlinear Problem with a Linear MILP Model," *Computer Aided Process Engineering*, Elsevier, pp. 647–652.

[10] L. PROLL, and B. SMITH, "Integer Linear Programming and Constraint Programming Approaches to a Template Design Problem," *INFORMS Journal on Computing*, vol. 10, no. 3, 1998, pp. 265-275.

[11] K.R. Apt, and M. Wallace, Constraint Logic Programming using Eclipse, Cambridge University Press New York, NY, USA, 2007.