

Reformulation during Automated Constraint Modelling

student: Andrea Rendl
supervisors: Ian Miguel and Ian P. Gent

School of Computer Science, University of St Andrews, UK
{andrea, ianm, ipg} @cs.st-andrews.ac.uk

Abstract. Constraint modelling requires considerable expertise and therefore automating this process is highly desirable to encourage the widespread use of constraint programming techniques. One approach is to refine a high-level problem specification into a solver-independent constraint model that is then tailored to a particular solver. We further investigate tailoring, the last step of the modelling process, and present reformulations to enhance the constraint model's quality.

1 Introduction

The process of formulating an *effective* constraint model is notoriously difficult, preventing the wider use of constraint solving. Hence, automated modelling is highly desirable. One approach is to provide an abstract problem specification that is then *refined* automatically into a constraint model. The ESSENCE abstract constraint specification language [1] and CONJURE automated refinement system [2] embody this approach. CONJURE produces constraint models in ESSENCE', a subset of ESSENCE, which has a level of abstraction similar to existing constraint solvers. ESSENCE' is solver-independent so a further 'tailoring' step is necessary to produce input for a particular solver. Tailoring a model efficiently to a solver is crucial to exploit that solver's individual features.

One way to improve model quality is through reformulation. In the automated modelling process described above, there are 3 levels at which reformulation can be applied:

- specification level (reformulating problems specified in ESSENCE)
- refinement level (i.e. during refinement of ESSENCE specifications into ESSENCE' models)
- tailoring level (i.e. when tailoring ESSENCE' models to a particular solver)

Thus far, we have investigated reformulations at the tailoring level and developed a translator [7] that tailors ESSENCE' models to the constraint solver MINION [3], which we summarise in this paper.

MINION Constraints	Meaning
<code>sumleq([x1,x2,...,xn], r)</code>	$x_1 + x_2 + \dots + x_n \leq r$
<code>sumgeq([x1,x2,...,xn], r)</code>	$x_1 + x_2 + \dots + x_n \geq r$
<code>weightedsumleq([x1,...,xn],[c1,...,cn], r)</code>	$x_1 * c_1 + \dots + x_n * c_n \leq r$
<code>weightedsumgeq([x1,...,xn],[c1,...,cn], r)</code>	$x_1 * c_1 + \dots + x_n * c_n \geq r$
<code>product(x1, x2, r)</code>	$x_1 * x_2 = r$
<code>eq(x1,x2)</code>	$x_1 = x_2$
<code>diseq(x1,x2)</code>	$x_1 \neq x_2$
<code>ineq(x1,x2,c)</code>	$x_1 \leq x_2 + c$
<code>max([x1,...,xn],r)</code>	$max(x_1, \dots, x_n) = r$
<code>min([x1,...,xn],r)</code>	$min(x_1, \dots, x_n) = r$
<code>element(v,i,r)</code>	$v[i] = r$
<code>alldiff(x1,...,xn)</code>	$x_1 \neq x_2 \neq \dots \neq x_n$
<code>reify(constraint,r)</code>	$if(constraint) \text{ then } r = 1 \text{ else } r = 0$
<code>table(matrix, tuple)</code>	an extensional constraint

Table 1. MINION constraints: \mathbf{x} and \mathbf{r} refer to decision variables, \mathbf{v} to a decision variable vector, and \mathbf{c} refers to a constant. Lists of decision variables $[x_1, \dots, x_n]$ may be replaced by vectors and rows or columns of matrices.

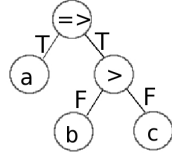
2 Tailoring ESSENCE' models to MINION

Tailoring ESSENCE' models to MINION instances is a challenging task: MINION provides a small set of constraints while ESSENCE' is an expressive language providing complex structures, such as nested quantification. In some cases, reformulations are essential to produce effective MINION models. MINION solves individual instances only. The input to the tailoring process is therefore an ESSENCE' model and parameter values for that model. MINION does not support nested constraints, so we need to decompose nested ESSENCE' expressions into simple MINION constraints (*flattening*, for an overview of basic MINION constraints see Table 1).

2.1 Arithmetic and Logical Constraints

Arithmetic Expressions If an arithmetic expression cannot be directly mapped to a corresponding MINION constraint, it is decomposed to a set of subexpressions by using auxiliary variables: each subexpression is constrained to be equal to an auxiliary variable, which is then used as a substitute for the subexpression (a simple example is given in Table 2). This approach can produce an overhead of auxiliary variables and subconstraints if the expressions are not mapped efficiently. Hence we make best use of MINION's constraint library, for example translating an iterated summation using the global `sum` constraint.

Basic Logical Expressions Boolean ESSENCE' expressions are translated using 0/1 variables and arithmetic constraints. Nested logical expressions are more difficult to flatten: unlike arithmetic expression flattening, this requires *reification*. Reification equates the satisfaction of some constraint c with a Boolean variable. Instead of assigning each subexpression to an auxiliary variable (as done with arithmetic expressions), each subexpression is reified to an auxiliary variable that again acts as its substitute in the super-expression (for an example see Table 2). MINION does not support reification of all its constraints, so during



Nested ESSENCE' Expr.	Corresponding MINION Constrs.
$(a * b) + c = d$	<code>product(a,b,t)</code> <code>sum([t,c], d)</code>
$a \Rightarrow (b > c)$	<code>reify(ineq(b,c,-1), t)</code> <code>ineq(t,a,0)</code>

Table 2. Flattening ESSENCE' expressions using the auxiliary variable t .

translation we need to be aware of the (sub)expression's *context*: if reification needs to be imposed on a subexpression, we can only employ reifiable MINION constraints. This is why we introduce a *reification flag* that indicates if the currently translated (sub)expression must be reified or not. The expression tree in Table 2 illustrates how the reification flag is set to *true* for the subexpressions of the implication, a and $b > c$, but becomes *false* for the subexpressions of the inequality, because it does not require reification.

Quantified Expressions Quantified expressions in ESSENCE' have the form

$$q \ i_1, \dots, i_n \in D. e(i_1, \dots, i_n)$$

where $q \in \{\forall, \exists\}$ is a quantifier, i_1, \dots, i_n are binding variables that range over the finite integer domain D and $e(i_1, \dots, i_n)$ is an arbitrary relational expression involving $i_1..i_n$. Translation of quantified expressions is further complicated by the fact that nesting is allowed, i.e. e may also contain quantified expressions.

First we consider the case when quantifications are not nested. Universal quantification and existential quantification can be treated as conjunction and disjunction, respectively. Conjunction can be split into separate constraints (presuming it is not nested). Consider setting every element of a vector v to zero. The expression $\forall_{i \in [1..n]}. (v[i] = 0)$ corresponds to the set of constraints $v[1] = 0, \dots, v[n] = 0$. Now consider $\exists_{i \in [1..n]}. (v[i] = 0)$, which corresponds to $v[1] = 0 \vee \dots \vee v[n] = 0$. Disjunction is flattened via the use of reification. Every disjunct is reified to an auxiliary variable and the sum over all auxiliary variables

ESSENCE' expression	MINION constraints		Aux. variables	
$\exists i_1..i_n \in D.(e)$	<code>{ reify(m(e(i1,..in)), xi) i1..in ∈ D }</code>	$k+1$	<code>x1..xk</code>	k
$\forall i_1..i_n \in D.(e)$ <i>reify = false</i>	<code>{ m(e(i1,..in)) i1..in ∈ D }</code>	0	none	0
$\forall i_1..i_n \in D.(e)$ <i>reify = true</i>	<code>{ reify(m(e(i1,..in)), xi) i1..in ∈ D }</code> <code>sumgeq([x1, ..xk], k)</code> <code>sumleq([x1, ..xk], k)</code>	$k+2$	<code>x1..xk</code>	k

Table 3. Overhead during translation of quantifications; $m : e \rightarrow c$ returns the MINION constraint corresponding to ESSENCE' expression e and $k = |D|^n$.

is set to be greater than or equal to 1. In our example, this would give the set of reified constraints: `reify(eq(v[0],0),x1) ... reify(eq(v[n],0),xn)` and the sum `sumgeq([x1,...xn], 1)`.

We now consider the case where quantified expressions are nested inside other expressions. In this case, reification is sometimes required. For example, if x is some constrained variable, the expression

$$(x = 1) \Rightarrow \forall_{i \in [1..5]}.(m[i] \neq i)$$

will require the universally quantified constraint to be reified to a variable r and then the constraint $(x = 1) \Rightarrow (r = 1)$ posted. This is a further example of the use of the reification flag as described in Section 2.1. Table 3 gives an overview of the auxiliary variable and constraint overhead during the flattening of nested quantified expressions.

2.2 Global Constraints

ESSENCE' provides some global constraints that can be directly mapped to corresponding MINION global constraints, or an equivalent set of constraints if no exact equivalent is available. However, some ESSENCE' expressions are reformulated to MINION global constraints, such as matrix indexing by decision variables using the *element* constraint. `element(matrix,index,elem)`, states that the element at position `index` of (one- or multidimensional) matrix `matrix` equals to (the assignment of) `elem` [6]. For example, the ESSENCE' expression $m[1,x] = n$ can be expressed by `element(row(m,1),x,n)`. Care must be taken with the index range, since ESSENCE' supports arbitrary matrix index ranges, while MINION's matrix indices always start with 0. If, in our example above, matrix m 's indices would range from $c..k$ where $c \neq 0$, we would need to adjust the index variable according to its offset c . This is achieved by subtracting the offset. For example, with m ranging from $1..k$, we obtain `element(row(m,0), x', n)` where $x' = x - 1$.

An interesting case of reformulation occurs if the matrix is entirely indexed by decision variables. Consider the Mutually Orthogonal Latin Squares (MOLS) problem [10]: a latin square is an $m \times m$ matrix of elements $1..m$ where each row and column has distinct elements. Two latin squares A and B are mutually orthogonal, if each pair of elements $(A(i,j), B(i,j))$ occurs exactly once, as illustrated in the example below. We can model this problem by introducing two auxiliary matrices X and Y that hold the row and column indices of A and B 's elements such that $A[X[i,j], Y[i,j]] = i$ and $B[X[i,j], Y[i,j]] = j$. If such matrices X, Y exist, then A and B are mutually orthogonal.

`element` is restricted to one index parameter, hence to translate an expression $A[x,y] = c$, we need to flatten matrix A to a vector A' where $A[i,j] = A'[i*r+j]$ and r corresponds to the amount of rows in matrix A . Hence we obtain the

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{pmatrix} \quad (A(i,j), B(i,j)) = \begin{pmatrix} (1,1) & (2,2) & (3,3) \\ (2,3) & (3,1) & (1,2) \\ (3,2) & (1,3) & (2,1) \end{pmatrix}$$

element constraint `element(A',i,c)` with an adjusted index $i = x * r + y$. Being able to index matrices with decision variables without the need to flatten them by hand is essential, because it allows to express further constraints that are more easily expressed over matrix structures: consider the MOLS problem again, where we need to impose *alldifferent* on all rows and columns of matrices A and B . This can be more easily expressed over a matrix structure than a flattened vector structure.

2.3 Low-level Choices

For efficiency reasons, MINION distinguishes several different types of variables and constraints, giving us choices we have to make.

Constraint selection In most cases the differences between constraint variants arise from the type of propagation, through the use of ‘watched literals’ [4]. Examples are `sum` and `element`. Watched literal-based constraints can reduce the search time drastically [4]. However, classical propagators can still be more effective when used appropriately, because they are cheaper to maintain. There is still no generalisation of when watched literals are sure to be more effective than classical propagators, but it is estimated that watched-literal based propagation performs better with loose constraints. We generally choose watched literal-based constraints, but give the user the possibility to force unwatched constraints to be applied by setting a flag. The current version of MINION does not support reification of watched literal-based constraints, so unwatched constraints are always chosen in a context where the reification flag is true.

Variable Selection When mapping bound variables from ESSENCE’ to MINION, we can choose between simple bound variables and discrete bound variables. Both are defined by an upper and lower bound, but discrete bound variables allow the removal of values between the bounds during search. This can be very effective, for instance in combination with the *alldifferent* constraint [5]. Not using discrete bound variables can even result in run-time errors on attempted value removals in MINION. This is why we generally apply discrete bounds variables, but allow the user to choose simple bound variables by a flag.

2.4 Simplifications

Simplifications of both the ESSENCE’ specifications and MINION model improve the resulting constraint model and also speed up the translation process.

Evaluation and Reduction Most evaluation and reduction steps are done during a preprocessing phase, where instances are generated by inserting parameter values which are then simplified. After insertion, we evaluate expressions consisting entirely of constants and apply simple Boolean axioms, such as evaluating $E \vee true$ to *true*. Note, that we can detect violations (expressions that are evaluated to *false*) during the preprocessing phase. We restrict evaluation to simple rules, because full evaluation would obviously

take more than polynomial time in general. After evaluation every constraint that has been evaluated to *true* is discarded.

Variable Reuse When a variable x is assigned to another variable y , such as in the ESSENCE' expression $y = x$, we can 'reuse' x by substituting x for any occurrence of y . This procedure can also be applied for (parts of) matrices, since in MINION decision variables can occur in several different matrices. Consider the example $\forall_{i \in 1..10}. v[i] = u[i + 1]$, where we assign every second element of vector u to vector v . Here we construct vector v out of the corresponding elements of u .

3 Conclusions, Related and Future Work

Our research concerns reformulation at three different points in the automated modelling process. Here we have summarised initial work at the lowest level: tailoring a solver-independent model to a particular constraint solver. The tailoring process may be compared to translating OPL to Ilog Solver, of which, to the best of our knowledge, no details have been published. Furthermore OPL lacks, for example, existential quantification which, when nested, significantly complicates the translation process, as we have seen. Charnley *et al* [8] describe a process of translating problems stated in first order logic to the Sicstus constraint solver. This is significantly easier than the translation process we have considered: the source language is less rich than ESSENCE' and the target language is significantly richer than MINION's input language. Rafah *et al.* present the mapping process of Zinc [9], a modelling language, to design models that apply different solving techniques rather than focusing on constraint solving alone.

In future work, we intend to build on the translator described here to incorporate more model-enhancing reformulations. We will also begin to explore reformulation of ESSENCE specifications and reformulation during refinement.

Acknowledgements Ian Miguel is supported by a UK Royal Academy of Engineering/EPSC Research Fellowship. Andrea Rendl is supported by a DOC FFORTE scholarship from the Austrian Academy of Sciences and UK EPSRC grant EP/D030145/1. We thank Chris Jefferson for his advice on MINION.

References

1. A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. *IJCAI*, pp80-87, 2007.
2. A.M. Frisch, C. Jefferson, B. Martínez Hernández, and I. Miguel. The rules of constraint modelling. *IJCAI*, pp 109–116, 2005.
3. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pp 98–102, 2006.
4. I.P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *CP*, pp 182–197, 2006.
5. J.-C. Régin. A filtering algorithm for constraints of difference in cps. *AAAI*, pp 362–367, 1994.
6. P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with ai and or techniques. *AAAI*, pp 660–664, 1988.
7. I.P. Gent, I. Miguel and A. Rendl. Tailoring Solver-independent Constraint Models: A Case Study with Essence' and Minion *SARA*, 2007. to appear
8. J. Charnley, S. Colton, and Ian Miguel. Automatic generation of implied constraints. *ECAI*, pp 73–77, 2006.
9. M. Garcia de la Banda, K. Marriott, R. Rafah, M. Wallace From Zinc to Design Model. *PADL*, LNCS 4354, pp.215-229, 2007.
10. W. Harvey and T. Winterer Solving the MOLR and Social Golfers Problems. *CP*, pp 286-300, 2005.