Tailoring Solver-Independent Constraint Models: A Case Study with Essence′ and Minion

Ian P. Gent, Ian Miguel, and Andrea Rendl

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

Abstract. In order to apply constraint programming to a particular domain, the problem must first be *modelled* as a constraint satisfaction problem. There are typically many alternative models of a given problem, and formulating an effective model requires a great deal of expertise. To reduce this bottleneck, the Essence language allows the specification of a problem *abstractly*, i.e. without making modelling decisions. This specification is refined automatically by the Conjure system to a solver-independent constraint modelling language Essence′. However, there is still significant work involved in translating an Essence′ model for use with a particular constraint solver. This paper discusses this 'tailoring' process with reference to the constraint solver Minion.

1 Introduction

Constraint programming is a successful technology for tackling a wide variety of combinatorial problems. To use constraint technology to solve a problem, the problem must first be described in terms of a *constraint model* suitable for input to a *constraint solver*, which then searches for solutions automatically. The process of formulating an *effective* constraint model (i.e. for which the intended constraint solver is able to find solutions efficiently) is notoriously difficult and is one of the major bottlenecks preventing the wider use of constraint solving.

Hence, automating constraint modelling is highly desirable. In one approach the user provides an abstract problem specification in which detailed modelling decisions have not yet been taken. This specification is then *refined* automatically into a constraint model. The Essence abstract constraint specification language [2] and Conjure automated refinement system [4] embody this approach. The constraint models produced by Conjure are in the language Essence′, which has a level of abstraction supported by existing constraint solvers.

Essence′ is, however, a solver-independent constraint language; Essence′ models must undergo a further translation step to produce input suitable for any particular constraint solver. The difficulty of this task depends on the facilities offered by the intended solver. Moreover, as shown by Prosser and Selenksy [10], individual constraint solvers have differing strengths and weaknesses. Therefore, an Essence′ model must be *tailored* to an individual solver to maximise efficiency. This paper considers the tailoring process, with particular reference to the constraint solver Minion [6]. This is a particular challenge, since Minion has been deliberately pared down to increase solving efficiency.

I. Miguel and W. Ruml (Eds.): SARA 2007, LNAI 4612, pp. 184–199, 2007.

2 Background

We begin by providing the necessary background in constraint modelling and solving before describing the Essence′ language and Minion constraint solver.

2.1 Constraint Satisfaction Problems and the Modelling Bottleneck

The finite-domain *constraint satisfaction problem* (CSP) consists of: a finite set of decision variables, $\mathcal{X}$; for each variable $x \in \mathcal{X}$, a finite set $\mathcal{D}(x)$ of values (its domain); and a finite set $\mathcal{C}$ of constraints on the variables, where each constraint $c \in \mathcal{C}$ is defined over a subset of $\{x_i, \ldots, x_j\}$ of $\mathcal{X}$ (its *scope*, denoted scope(c)) by a subset of the Cartesian product $\mathcal{D}(x_i) \times \cdots \times \mathcal{D}(x_j)$ giving the set of allowed combinations of values. A constraint can be specified extensionally, by listing these tuples, or intensionally as an expression with an associated *propagation* algorithm that is executed by the constraint solver to determine satisfaction/violation. A solution to a CSP assigns values to all variables such that all constraints are satisfied.

A constraint model maps the features of a combinatorial problem onto the features of a CSP. The CSP is input to a constraint solver, which searches for a solution (or solutions). The constraint model is then used to map the solution(s) back onto the original problem. Constraint languages and solvers commonly provide a rich library of constraints from which to choose. Typically, therefore, many models are possible for a given problem. This choice is both complex and important: it can mean the difference between the problem being solved quickly and not being solvable in a practical amount of time. Hence, constraint modelling is difficult and requires a great deal of expertise.

The constraint specification language Essence [2] addresses this modelling bottleneck. It enables the specification of a combinatorial problem *abstractly*, without making constraint modelling decisions. This is achieved by supporting decision variables whose domain elements are the combinatorial objects (e.g. functions, relations) that a combinatorial problem commonly requires us to find, but which are not directly supported by existing constraint solvers.

To illustrate, consider Langford's problem (CSPlib problem 24), which is to arrange n sets of positive integers $1..k$ into a sequence such that, following the first occurrence of an integer i, each subsequent occurrence of i appears $i + 1$ indices later than the last. For example, if k is 4 and n is 2, then a solution is 41312432. The problem may be viewed as requiring us to find a bijection from the values to their positions in the sequence. This can be stated directly in Essence, as Figure 1 shows. For simplicity, we set $n = 2$, hence the sequence

`given`	k : `int`(1 ...)
`find`	*positions* : `int`$(1..2 * k) \rightarrow$ (`bijective`) `int`$(1..2 * k)$
`such that`	`forall` i : `int`$(1..k)$. *positions*$(i + k) -$ *positions*$(i) = i + 1$

Fig. 1. An Essence specification of a simplified version of Langford's problem

has length $2k$. We use $2k$ values to represent the elements of the sequence. The first occurrence of integer i is represented by i itself, and the second occurrence by $i+k$. The key feature of this specification is that it contains just one decision variable, *positions*, whose domain is the set of possible bijections from $1..2k$ onto $1..2k$. The single constraint ensures that the pairs of elements are the requisite number of indices apart. The bijection in this specification can be modelled in a variety of ways [9], but ESSENCE does not force the user to make this decision.

2.2 The ESSENCE′ Solver-Independent Modelling Language

Given an ESSENCE specification, constraint modelling consists in encoding the abstract decision variables and the constraints on them as constrained collections of CSP variables. The CONJURE [4] automated refinement system performs this step automatically, producing a constraint model in a subset of ESSENCE called ESSENCE′. ESSENCE′ is a solver-independent constraint modelling language with a level of abstraction that is supported by existing constraint solvers. ESSENCE′ is intended to be very nearly an object-level language. However, as we demonstrate in this paper, there remain issues in translating to a specific language that must be resolved carefully to avoid affecting performance adversely.

Decision variables are specified individually or as elements of multi-dimensional matrices. Their domains may contain integers or Booleans only, in contrast with the abstract decision variables offered by ESSENCE. The available constraint library includes basic building blocks, such as the ability to specify constraints extensionally, and arithmetic and logical operators, which can be nested to form arbitrarily complex constraint expressions. It also includes commonly-used intensional constraints such as the all-different constraint [11], which constrains a matrix of variables to take distinct values, and the lexicographic ordering constraint [3], which can be used to deal with symmetry in constraint models. Existential and Universal quantifiers are also supported and may be nested.

To illustrate, Figure 2 shows an ESSENCE′ model of the simplified Langford's problem specified in Figure 1. The refinement here is simple: the function is modelled using a matrix of decision variables indexed by the values in the sequence. The domain of each decision variable is the set of possible positions. Note the use of the all-different constraint to ensure that the bijective property holds.

```
1  given k : int(1..)
2  find positions : matrix indexed by [int(1..2*k)] of int(1..2*k)
3  such that
4   forall i : int(1..k) .
5     positions[i+k] = positions[i] + i+1,
6   allDifferent(positions)
```

Fig. 2. ESSENCE′ model of a simplified version of Langford's problem

2.3 The MINION Constraint Solver

The MINION [6] constraint solver is designed to promote solving efficiency. Consequently, its input language is relatively restricted, but it does offer the option to tune various low-level features that are not commonly accessible.

Like ESSENCE′, MINION supports decision variables with integer domains, which can be collected in multi-dimensional matrices. It also provides fine control over domain representation. MINION supports four individually-optimised integer domain types: 0/1 domains, commonly used for logical expressions and counting; *bounds* domains, which maintain only the lower and upper bound of the domain; *sparse* domains, where domain values need not be adjacent, e.g. {2, 7, 11}; and *discrete* domains, which are defined initially by lower and upper bounds but which support the removal of values from between the bounds.

Table 1. MINION constraints: `x` and `r` refer to decision variables, `v` to a decision variable vector, and `c` refers to a constant. Lists of decision variables `[x1,..,xn]` may be replaced by vectors and rows or columns of matrices.

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

```
1  0
2  8
3  1 8 8
4  0
5  0
   .
   .
   .
6  1
7  [x0,x1,x2,x3,x4, x5, x6, x7],
   .
   .
   .
```

```
   .
   .
   .
8  sumgeq([x0, 2], x4)
9  sumleq([x0, 2], x4)
10 sumgeq([x1, 3], x5)
11 sumleq([x1, 3], x5)
12 sumgeq([x2, 4], x6)
13 sumleq([x2, 4], x6)
14 sumleq([x3, 5], x7)
15 sumleq([x3, 5], x7)
16 alldiff(v0)
```

Fig. 3. Partial MINION instance of the simplified Langford's problem ($k = 4$)

MINION offers a broadly similar set of constraints (summarised in Table 1) compared with ESSENCE′, but again provides controls as to how they are implemented, as will be explained below. A key difference is that MINION supports neither quantification nor nested constraint expressions. It also allows the statement of individual *instances* only, rather than parameterised problem classes.

To illustrate, Figure 3 presents part of the MINION input file for the instance of Langford's problem when $k = 4$. The MINION manual [8] contains full details of the input format. Briefly, in a model with n variables, each variable is identified by 'xi', for i in $0..(n-1)$. MINION insists that variables of each type described above are declared in turn. In the example there are 8 bounds variables with lower bound 1 and upper bound 8 (lines 2-3). These variables are collected in the vector $v0$ (lines 6-7). The single universally-quantified constraint from the ESSENCE′ model is expanded into a set of individual sum constraints (lines 8-15). Note that MINION requires that each equality constraint is decomposed into a pair of sum constraints. The all-different constraint is added directly (line 16).

3 Tailoring ESSENCE′ to MINION: Overview

Since MINION expects individual instances, the input to the tailoring process is a pair: an ESSENCE′ model, and a set of values for the parameters of the model sufficient to determine a particular instance. The first (straightforward) step in the tailoring process is therefore to parse the ESSENCE′ model and, for each parameter, substitute its given value for each of its occurrences in the model. Following substitution, a simplification step is performed to evaluate expressions now composed entirely of constants. These pre-processing steps are independent of the target solver.

In translating ESSENCE′ to MINION, we make use of a `MinionModel` structure, a four-tuple $\langle V, A, C, M \rangle$ where: V is the set of MINION decision variables; A is the set of MINION matrices, whose elements are drawn from V; C is the set of MINION constraints; and M is a bijection between the elements of V and the original ESSENCE′ variables. The function M is important both for mapping solutions produced by MINION back onto the ESSENCE′ model and to avoid the repeated introduction of MINION variables for a single ESSENCE′ variable that occurs in several constraints.

We define a translation function τ as follows:

$$\tau : \langle \mu, e \rangle \rightarrow \mu'$$

where μ is an instance of a `MinionModel` and e is an ESSENCE′ expression. This pair is mapped to a new `MinionModel` instance μ'. The effect of applying τ is to increase monotonically the four elements of the `MinionModel`. Beginning with an empty `MinionModel`, tailoring of an ESSENCE′ model proceeds through the constraint-wise application of τ, incrementally constructing the final model.

Since ESSENCE′ is the richer language, typically each ESSENCE′ constraint corresponds to a set of MINION constraints. Variables are introduced into the `MinionModel` both to correspond to the variables in the original ESSENCE′ model (in which case the correspondence is recorded in the mapping M) and to support the decomposition of an ESSENCE′ expression. Decomposition is necessary to deal both with nested expressions and with the quantifiers and constraints not directly supported by MINION, as is described in the following sections.

4 Arithmetic Constraints

We begin by discussing the translation of arithmetic expressions. As noted, MINION provides the small set of constraints given in Table 1. ESSENCE′ constraints of exactly this form require no translation. For the remainder, simple *reformulation* steps are performed. Note, for example, that MINION does not provide a division operator. Hence, division in ESSENCE′ is translated by rewriting division to multiplication. In general an ESSENCE′ arithmetic expression is translated via combinations of the primitive constraints in Table 1, sometimes connected by introducing auxiliary variables. A very simple example can be seen in the MINION instance of Langford's problem (Figure 3): to constrain a sum of variables and constants to be equal to a variable or a constant, a pair of `sumgeq` and `sumleq` constraints is used (e.g. lines 8-9). Equality of weighted sums is treated similarly.

Commonly, ESSENCE′ arithmetic constraints are translated by *flattening* nested expressions. That is, an ESSENCE′ expression is decomposed into sub-expressions for which MINION provides a corresponding constraint. All MINION constraints used for this purpose are expressed in terms of equality or inequality relations, as per the examples in Table 1. Hence, an auxiliary variable is constrained to be equal to each sub-expression. Constraints among the auxiliary variables are added as necessary to create a set of MINION constraints equivalent (i.e. collectively allowing the same set of assignments) to the original ESSENCE′ constraint.

To illustrate, consider an ESSENCE′ constraint that constrains the sum of n variables to be not equal to some variable r. MINION provides a binary disequality constraint, but no direct way to express a disequality on a sum. Hence, it is natural to introduce an auxiliary variable constrained, in the same way as above, to be equal to the sum of the n variables *and* to be not equal to r:

ESSENCE′	MINION

$x_1 + \ldots + x_n \neq r$	`sumleq([x1,x2,...,xn], a)`
	`sumgeq([x1,x2,...,xn], a)`
	`diseq(a, r)`

Equality and disequality of weighted sums are treated similarly.

When an auxiliary variable is introduced it must be given an appropriate domain. In doing so, we can exploit our knowledge of the MINION solver. Propagation of each of the constraints in Table 1 affects only the bounds of the variable `r`. Hence, it is sufficient to use an efficient bounds domain (see Section 2.3) for each auxiliary variable introduced in translating an arithmetic expression. The lower and upper bound of the domain of an auxiliary variable is determined by examining the lower and upper bounds of the expression to which it is constrained to be equal. Hence, in the current example the domain of a ranges from the sum of the lower bounds of $x_1, ..., x_n$ to the sum of their upper bounds.

Our general method of translating nested ESSENCE′ expressions proceeds from the parse tree as follows:

1. Consider the nested Essence′ expression a tree with variables and constants as leaves, operators as nodes and the tree branched according to the operator precedences where the operator with lowest precedence is root.
2. Take an operator node *op* with highest depth that connects leaves $l_1..l_n$. Generate the corresponding Minion constraint(s) for $op(l_1, .., l_n) = v$ where v is a auxiliary variable. Add the generated constraints to C and v to V.
3. If the tree contains at least another leaf, replace node *op* by leaf v and go to 2., otherwise stop.

To minimise the generated overhead of variables and constraints during flattening, we can apply simple and effective rules: directly match Minion primitives to the expression tree structure, such as (weighted) sums or products. With a subtree matching an iterated sum structure with n leaves, this strategy reduces the number of auxiliary variables from $n-1$ (or n for the whole tree) to 1.

5 Logical Constraints

Boolean Essence′ expressions are translated using 0/1 variables and arithmetic constraints. Table 2 summarises how the common logical connectives are translated for Minion. As in the arithmetic case, flattening is required for nested logical expressions. This operates in broadly the same way as for arithmetic, but differs in that it requires *reification*. Reification can be viewed as a 'meta' constraint in that it equates the satisfaction of some constraint c with a Boolean variable. Minion provides reification using 0/1 variables and the constraint: `reify(c, x1)`, meaning `x1` is assigned 1 if and only if `c` is satisfied.

To illustrate, consider the Essence′ expression $(x1 = x2) \Rightarrow (x3 = 0)$. Minion does not provide a constraint of this form, so flattening is required. To do so, we decompose the implication into left- and right-hand sides and reify them into two auxiliary 0/1 variables: `reify(eq(x1, x2), a1)`, `reify(eq(x3, 0), a2)`. The implication can now be stated using an inequality: `ineq(a1, a2, 0)`.

Not all constraints in Minion can be reified, hence care must be taken to employ only reifiable constraints for a subexpression if it will be reified. In order

Table 2. Basic logical connectives and their equivalents using Minion 0/1 variables. '`n`' is a unary operator on a Minion 0/1 variable `x` that returns 1−`x`. Clearly, the sum constraint is only necessary to express conjunction arising in some nested subexpression. Otherwise e_1 and e_2 can simply be imposed separately, since a solution requires all constraints to be satisfied. '`ineq(x1, x2, c)`' is interpreted $x_1 \leq x_2 + c$. If the e_i are nested expressions, flattening is required.

Logical Expression	Minion Constraint
$\neg e$	`nx`
$e_1 \wedge e_2$	`sumgeq([x1, x2], 2)`
$e_1 \vee e_2$	`sumgeq([x1, x2], 1)`
$e_1 => e_2$	`ineq(x1, x2, 0)`
$e_1 <=> e_2$	`eq(x1, x2)`

to determine if a subexpression will be reified or not, we need to have information about its *context*. This is why we introduce a *reification flag*, that indicates when *true* that the currently translated expression will to be reified. The reification flag is initially false, retains its state as we traverse certain constraints (e.g. a conjunction) but becomes true when we traverse other constraints (e.g. a disjunction).

An interesting case of reformulation occurs with negation: Essence′ supports negation of logical expressions while Minion only allows the negation of single variables (see Table 2). This is why negation is applied to expressions before translation. Negated relational expressions are reformulated by applying the corresponding complementary operator, for example $\neg(x = y)$ is reformulated to $x \neq y$. Negated Boolean expressions are reformulated by applying Boolean axioms: $\neg(x \wedge y)$ results in $\neg x \vee \neg y$. In the last case the negation operator is passed down a level in the expression tree, eventually being applied to a variable or relational expression. Hence, the reformulation process halts, even though the worst case may take exponential time in the depth of the expression tree.

Generally, flattening of logical expressions proceeds directly from the parse tree, as described in Section 4, but making use of reification rather than arithmetic constraints for decomposition. However, care must be taken in the presence of universal and/or existential quantification. Quantified expressions in Essence′ have the form $q \;\; i_1, \ldots, i_n \in D.\ e(i_1, \ldots, i_n)$ where $q \in \{\forall, \exists\}$ is a quantifier, $i_1, \ldots i_n$ are binding variables that range over the finite integer domain D and $e(i_1, \ldots, i_n)$ 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. In what follows, we describe a general, effective approach to translating arbitrarily-nested quantified Essence′ expressions into Minion.

5.1 Singly-Quantified Expressions

We first define a basic approach for translating quantified expressions. Universal quantification can be treated as a conjunction, existential quantification as a disjunction of expressions. Consider the example $\forall_{i \in [1..5]}.(m[i] \neq i)$. It corresponds to the conjunction $(m[1] \neq 1) \wedge .. \wedge (m[5] \neq 5)$, and can be translated to 5 separate constraints. Now consider $\exists_{i \in [1..5]}.(m[i] \neq i)$ that corresponds to a disjunction $(m[1] \neq 1) \vee .. \vee (m[5] \neq 5)$. In this case we need to apply reification. Imposing `reify(diseq(m[1],1), x1)` gives us a reified variable `x1` that is set to 1 if `diseq(m[1],1)` holds and 0 if not. So the disjunction is satisfied, if at least one reified variable equals to 1. We can enforce that by imposing another constraint, stating that the maximum of all reified variables has to equal 1: `sumgeq([x1,x2,x3,x4,x5],1)`. Conjunction can be translated by insisting that the sum of the k reified variables equals k. Thus the translation of disjunction introduces n auxiliary variables and $n + 1$ reification constraints with $n =\mid D \mid$ where D is the domain of binding variable i. Hence, in the general case with m binding variables over domain D, we get n^m additional variables, $n^m + 1$ reification constraints, and n^m expressions to translate.

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

ESSENCE′ expression	MINION constraints	Auxiliary variables
$\exists i_1..i_n \in D.(e)$	{ `reify(` $m(e(i_1,..i_n))$`, xi)` $\mid i_1..i_n \in D$ } `sumgeq([x1, ..xk], 1)`	`x1..xk`
$\forall i_1..i_n \in D.(e)$ $reify = false$	{ $m(e(i_1,..i_n))$ $\mid i_1..i_n \in D$ }	none
$\forall i_1..i_n \in D.(e)$ $reify = true$	{ `reify(` $m(e(i_1,..i_n))$`, xi)` $\mid i_1..i_n \in D$ } `sumgeq([x1, ..xk], k)` `sumleq([x1, ..xk], k)`	`x1..xk`

5.2 Nested Quantification

While existential quantification, as a form of disjunction, can only be translated using reification, universal quantification is a form of conjunction and sometimes can be translated without reification. We saw above that the expression $\forall_{i \in [1..5]}.(m[i] \neq i)$ can be translated without reification. Generally, a translation without reification is to be preferred as being simpler and allowing propagation more easily. Unfortunately, reification is sometimes necessary where universally quantified constraints are nested. 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 universal constraint to be reified to a variable r and then the constraint $x = 1 \Rightarrow r = 1$ posted. This shows another application of the reification flag.

Generally, we translate quantifiers by inserting values for their binding variables and translate the resulting expressions according to the imposed quantifiers. We apply values for binding variables recursively, starting with the outmost quantifier, and then stepwise increase the values, beginning with the first variable of the innermost quantifier. If a variable has reached its upper bound, it is reset to its lower bound and the next variable's value is increased. When the last binding variable of the outmost quantifier has reached its upper bound, we stop. During this value-insertion process we build the resulting constraints from the inside out: Depending on the quantifier and the reification flag, we generate a set of constraints. Consider the quantification $\exists_{i \in [0..5]} \forall_{j \in [1..3]}.m[j] \neq i$, that corresponds to a disjunction of conjunctions of $m[j] \neq i$, as shown below.

$$\begin{aligned} \exists_{i \in [0..5]} \forall_{j \in [1..3]}.(m[j] \neq i) = \\ & \bigvee m[1] \neq 0 \wedge m[2] \neq 0 \wedge m[3] \neq 0 \\ & \bigvee m[1] \neq 1 \wedge m[2] \neq 1 \wedge m[3] \neq 1 \\ & \bigvee ... \\ & \bigvee m[1] \neq 5 \wedge m[2] \neq 5 \wedge m[3] \neq 5 \end{aligned}$$

The conjoined expressions $m[j] \neq i$ are in a disjunctive context because of the outermost existential, so we set the reification flag to be *true*. First we insert lower bounds of variables i, j and then increase j's value until we reach its upper bound, getting the set of expressions $m[1] \neq 0, m[2] \neq 0, m[3] \neq 0$. We impose the

innermost quantifier, $\forall$, giving us a conjunction to translate: $m[1] \neq 0 \wedge m[2] \neq 0 \wedge m[3] \neq 0$. With the reification flag being *true*, we have to reify the expressions and get 3 reification constraints `reify(`$m[1] \neq 0$`,x1) ... reify(`$m[3] \neq 0$`,x3)` and a sum constraint `sumgeq([x1,x2,x3], 3)`, introducing 3 auxiliary variables `x1, x2, x3`. Since i can take 6 different values, we have 6 conjunctions to translate, resulting to 18 reification constraints and 6 sum constraints. We now impose the $\exists$ quantifier on our set of conjunctions. Each conjunction is represented by a sum constraint that we reify: `reify(sumgeq([x1,x2,x3], 3), x00), .. reify( sumgeq([x16,x17,x18], 3), x05)` and express disjunction by the sum `sumgeq([x00,x01,x02,x03,x04,x05],1)` over the auxiliary variables `x00 .. x05`, according to Table 3.

5.3 Treating Special Cases

These rules are all applied in the basic implementation. They allow us to translate expression "on the fly", since it can be immediately determined which rule to apply and the translation does not depend on future events (the translation is causal). Consequently, we only employ a small amount of memory during translation. However, there are cases where we translate quantified expressions that are later shown to be redundant. Consider the example $\exists_{i \in [1..n]}.e \vee i = n$ where e is an arbitrary expression. Since n is in the range of i, the expression will evaluate to *true*. The existential quantification, corresponding to disjunction, also becomes *true*. Please note, that this case may only be spotted at instance level, since the domain of a binding variable may depend on a parameter value. To detect cases where quantified expressions are always satisfied or violated requires significant effort. We will investigate this procedure in future, but are aware that the overhead might outweigh the benefit gained.

6 Global Constraints

Global Constraints [13] represent general problem patterns, e.g. that every element of a datastructure has a distinct value is captured by the global constraint *alldifferent* [11]. Constraint solvers usually provide efficient propagators for global constraints. Essence′ provides a range of global constraints which can be directly mapped to the corresponding Minion global constraint, or an equivalent set of constraints added if no exact equivalent is available.

Minion provides different versions of some constraints, giving us choices we have to make. In most cases the differences arise from the type of propagation, through the use 'watched literals', a recently-introduced method for writing constraint propagators [7]. Examples are `sum` and `element`. Watched literal-based constraints can reduce the search time drastically [7]. However, classical propagators can still be more effective in some cases. 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.

MINION provides matrix indexing by decision variables using the *element* constraint, `element(matrix,index,elem)`, stating that the element at position `index` of (one- or multidimensional) matrix `matrix` equals to (the assignment of) `elem` [12]. We use this to translate matrix indexing in ESSENCE′. For example, the simple expression $m[1, x] = n$ can be expressed by `element(row(m,1),x,n)`. If dynamically indexed matrices occur in nested expressions, we need to introduce an auxiliary variable to represent the indexed matrix element. Furthermore, we are able to nest dynamic indexing and express a $k - 1$ times nested indexing with k element constraints. For instance, $m[1, v[x]] = n$, which is nested once, can be reformulated to 2 element constraints, `element(row(m,1), tmp, n)` and `element(v, x, tmp)` by introducing the auxiliary variable `tmp`.

Matrices that are entirely indexed by decision variables need to be flattened. We illustrate this case by considering the Mutually Orthogonal Latin Squares (MOLS) problem [15]: 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 model this problem by introducing two auxiliary matrices X and Y, holding 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 translating an expression $A[x, y] = c$ requires flattening 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 element constraint `element(A',i,c)` with an adjusted index $i = x*r+y$. Indexing 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.

7 Variable Translation

As mentioned in Section 2.3, MINION has different types of integer decision variables: bounds- and discrete-domain variables. Discrete domain variables allow deletion of values inside the bounds during search, which can be very effective, for instance in combination with the *alldifferent* constraint. Where constraints such as this are used, use of bounds-domain variables in MINION risks run-time errors on attempted value removals. In ESSENCE′ there is no such distinction of bound variables, and so we generally translate decision variables to discrete-domain variables. We allow the user to select bounds-domain variables for translation by a flag if they are confident that discrete domains are unnecessary.

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

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.

8 Experimental Results

We have implemented a translator from ESSENCE′ to MINION in Java 1.5.0_10. In this section we report results using MINION on translated instances. Our translator is under active development and can be found at `minion.sourceforge.net`.

In this section we present some problems that we have specified in ESSENCE′ and tailored to a set of MINION instances. We compare these instances with optimised benchmarks produced by problem-specific instance generators provided by MINION. These generators have been implemented by experts in both constraint modelling and MINION. Hence we are comparing automatically tailored instances with human expertise. We performed our experiments on a 3GHZ Pentium 4 with 2GB RAM using MINION version 0.4.1 compiled with g++ 4.0.2 under Linux. The translator takes a neglegible amount of time for translation, hence we do not include its runtime and focus on how competitive the resulting instances are.

8.1 The Balanced Incomplete Block Design (BIBD)

The BIBD (CSPlib problem 28) is defined by a 5-tuple of positive integers $\langle v, b, r, k, \lambda \rangle$: assign v objects to b blocks such that each block contains k different objects and exactly r objects occur in each block and every two distinct objects occur in exactly λ blocks. A typical problem model consists of a 0/1 matrix m with b columns (blocks) and v rows (objects), where an element m_{ij} is assigned 1 if block i contains object j. Each row is constrained to the sum r and each column to the sum k. The scalar product of each two rows corresponds to λ. We order rows and columns lexicographically to break symmetry partially.

As summarised in Table 4, we compared generated MINION instances with BIBD instances generated by the hand coded BIBD-instance generator provided by MINION, with constraints using watched literals and without. Our translator produced almost identical instances to those produced by the instance-generator,

Table 4. Results for solving the BIBD problem

	Unwatched				Watched			
	Time(sec)		Nodes		Time(sec)		Nodes	
b, v, r, k, λ	Gen.	Trans.	Gen.	Trans.	Gen.	Trans.	Gen.	Trans.
140,7,60,3,20	0.51	0.51	17235	17235	0.67	0.7	17235	17235
210,7,90,3,30	2.08	2.04	67040	67040	3.03	3.07	67040	67040
280,7,120,3,40	6.73	6.51	182970	182970	9.45	9.34	182970	182970
315,7,135,4,45	10.73	10.67	278310	278310	15.14	15.32	278310	278310

Table 5. Results for finding a solution to the n-Queens problem

	Time(sec)			Nodes		
	Generator	Translator		Generator	Translator	
n	discrete var.	discrete var.	bounds var.	discrete var.	discrete var.	bounds var.
12	< 0.01	< 0.01	< 0.01	60	59	1,840
15	< 0.01	< 0.01	0.02	249	248	12,687
17	0.01	0.02	0.13	1,187	1,186	62,382
19	< 0.01	0.01	0.07	583	582	34,595
20	0.48	0.65	5.63	37331	37,330	2,857,524
22	3.84	5.19	55.74	269,370	269,369	28,458,527
24	1.01	1.34	15.19	63,791	63,790	7,528,769
25	0.12	0.16	2.08	7,272	7,271	943,172
26	0.96	1.27	16.97	55,592	55,591	8,057,222
27	1.16	1.56	20.47	67,231	67,230	9,723,687
29	3.94	5.09	72.69	212,276	212,275	35,867,550
30	141.24	193.57	>45min	7,472,996	7,472,995	1,379,220,754

performing almost exactly the same in means of time and identically in amount of search nodes used. This demonstrates the effectiveness of the tailoring process for the BIBD problem.

8.2 The n-Queens Problem

The n queens problem is to place n queens on a $n \times n$ chess board without attacking each other. In the problem model, the queens are represented by a vector v of length n where the element v_i corresponds to the column of the queen placed in row i. No queen may be placed on the same diagonal as another, which is specified using two auxiliary vectors of same length n. Each element of the auxiliary vectors has a distinct domain, which cannot be encoded in Essence′ and therefore has to be restricted by additional constraints. We produce instances with both bound-domain and discrete-domain variables and compare them to instances generated by an n-queens instance-generator for the same model provided wth the Minion distribution. This generator creates instances with discrete-domain variables and distinct bounds for the auxiliary variables. Results are given in Table 5. We see that the run times are notably slower than the hand-written generator, up to just under 40% in the largest instance. Nodes searched is always exactly one less. In general terms, these results show that we produce good models, but unsurprisingly there can be scope for further optimisation if writing a specialised generator. We can also observe the drawback of bound-domain variables with this problem.

8.3 The Quasigroup Problem

An m order quasigroup is an $m \times m$ multiplication table of integers $1..m$, where each element occurrs exactly once in each row and column and certain multiplication axoims hold. The quasigroup problem (CSPLib problem 3) is concerned with the existence of such a group of order m. We compared instances of the generator and translator with and without a special variable ordering. Variable orderings have been added by hand after the translation process since Essence′ has, at the time of writing, no facilities to specify variable orderings. Both instances

Table 6. Results for solving the QuasiGroup7 existence problem of order m

	with Ordering				without Ordering			
	Time(sec)		Nodes		Time(sec)		Nodes	
m	Trans.	Gen.	Trans.	Gen.	Trans.	Gen.	Trans.	Gen.
7	< 0.01	< 0.01	756	844	0.03	0.03	3,275	3,272
8	0.1	0.1	11,949	12,450	1.94	1.89	171,203	169,078
9	< 0.01	< 0.01	238	233	5.15	5.23	458,062	454,512
10	249.6	250.02	30,549,274	31,383,717	>1h	>1h	-	-

apply the same specified ordering. As demonstrated by the results in Table 6, an efficient variable ordering is crucial for solving the problem efficiently. Such an ordering can easily be added by the user in the MINION instance. Our tailored instances have shown a slightly better performance with variable orderings and are therefore highly competitive to the generated ones.

8.4 Summary

Our experimental results are not extensive, but show that instances tailored by our translator tend to perform well in comparison with those produced by instance generators implemented by modelling experts. Developing a generator takes significantly more time and knowledge about MINION than simply expressing a problem in ESSENCE′. We see in one case that the hand-coded generator runs faster, showing that (as expected) we cannot always attain optimal encodings automatically. Finally, our quasigroup results raise an important issue. Specifying good heuristics is often not considered to be a part of modelling but is well known to be essential to success. The more successful tools such as CONJURE and our translator become, the more important it will be to specify heuristics during this process, either manually or automatically.

There are a number of outstanding issues. The most important of these is that the only global constraints supported are alldifferent and element. To correct this is trivial for global constraints supported by Minion (e.g. table) but requires implementing an encoding of constraints which are not supported directly (e.g. global cardinality).

9 Conclusion

This paper has discussed the issues arising in tailoring models in a solver-independent constraint language, ESSENCE′, to the constraint solver MINION. This process may be compared with that of translating OPL to Ilog Solver, which formed part of a commercial product from Ilog. However, to the best of our knowledge, details of the OPL to Solver translation are unpublished. Furthermore OPL lacks, for example, existential quantification which, when nested, significantly complicates the translation process, as we have seen. Charnley *et al* [1] 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 because the source language is less rich than ESSENCE′ and

the target language is significantly richer than the input language of Minion. Rafeh *et al.* present the mapping process of Zinc [14], a modelling language, to design models that apply different solving techniques. Though Essence′ and Zinc are both on the same level of abstraction, the mapping targets are quite diverse. We would like to incorporate further reformulations into the tailoring process to enhance the final model. We will draw on our work on Cgrass [5] for this, which focused on the reformulation of individual problem instances.

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

References

1. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: European Conference on Artificial Intelligence (ECAI), pp. 73–77 (2006)
2. Frisch, A.M., Grum, M., Jefferson, C., Martínez Hernández, B., Miguel, I.: The design of essence: A constraint language for specifying combinatorial problems. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 80–87 (2007)
3. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artificial Intelligence 170(10), 803–834 (2006)
4. Frisch, A.M., Jefferson, C., Martínez Hernández, B., Miguel, I.: The rules of constraint modelling. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 109–116 (2005)
5. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: A system for transforming constraint satisfaction problems. In: International Workshop on Constraint Solving and Constraint Logic Programming, pp. 15–30 (2002)
6. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: European Conference on Artificial Intelligence (ECAI), pp. 98–102 (2006)
7. Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 182–197. Springer, Heidelberg (2006)
8. Gent, I.P., Jefferson, C.A., Miguel, I., Petrie, K., Rendl, A.: Minion manual, version 0.4.1., `http://minion.sourceforge.net`
9. Hnich, B., Walsh, T., Smith, B.M.: Dual modelling of permutation and injection problems. Journal of Artificial Intelligence Research (JAIR) 21, 357–391 (2004)
10. Prosser, P., Selensky, E.: A study of encodings of constraint satisfaction problems with 0/1 variables. In: International Workshop on Constraint Solving and Constraint Logic Programming, pp. 121–131 (2002)
11. Régin, J.-C.: A filtering algorithm for constraints of difference in csps. In: National Conference on Artificial Intelligence (AAAI), pp. 362–367 (1994)
12. Van Hentenryck, P., Carillon, J.-P.: Generality versus specificity: An experience with AI and OR techniques. In: National Conference on Artificial Intelligence (AAAI), pp. 660–664 (1988)

13. van Hoeve, W.-J., Katriel, I.: Global constraints. In: Handbook of constraint programming, Elsevier (2006)
14. de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: From Zinc to Design Model. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 215–229. Springer, Heidelberg (2006)
15. Harvey, W., Winterer, T.: Solving the MOLR and Social Golfers Problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 286–300. Springer, Heidelberg (2005)