Extensible Automated Constraint Modelling

Ozgur Akgun
ozgur.akgun@st-andrews.ac.uk
University of St. Andrews

Ian Miguel
ianm@cs.st-andrews.ac.uk
University of St. Andrews

Chris Jefferson
caj@cs.st-andrews.ac.uk
University of St. Andrews

Alan M. Frisch
frisch@cs.york.ac.uk
University of York

Brahim Hnich
brahim.hnich@ieu.edu.tr
Izmir University of Economics

Abstract

In constraint solving, a critical bottleneck is the formulation of an effective constraint model of a given problem. The CONJURE system described in this paper, a substantial step forward over prototype versions of CONJURE previously reported, makes a valuable contribution to the automation of constraint modelling by automatically producing constraint models from their specifications in the abstract constraint specification language ESSENCE. A set of rules is used to *refine* an abstract specification into a concrete constraint model. We demonstrate that this set of rules is readily extensible to increase the space of possible constraint models CONJURE can produce. Our empirical results confirm that CONJURE can reproduce successfully the kernels of the constraint models of 32 benchmark problems found in the literature.

Introduction

This paper attacks the problem of overcoming the *modelling bottleneck* through automating the constraint modelling process. In constraint solving, a critical bottleneck is the formulation of an effective constraint model of a given problem. This is considered to be one of the key challenges facing the constraints field (Puget 2004), and one of the principal obstacles preventing widespread adoption of constraint solving. Without help, it is very difficult for a novice user to formulate an effective (or even correct) model of a given problem. Recently, a variety of approaches have been taken to automate aspects of constraint modelling, including: machine learning (Bessiere et al. 2006); case-based reasoning (Little et al. 2003); theorem proving (Charnley, Colton, and Miguel 2006); automated transformation of medium-level solver-independent constraint models (Rendl 2010; Nethercote et al. 2007; Van Hentenryck 1999; Mills et al. 1999); and refinement of abstract constraint specifications (Frisch et al. 2005a) in languages such as ESRA (Flener, Pearson, and Ågren 2003), ESSENCE (Frisch et al. 2008), $\mathcal{F}$ (Hnich 2003) or Zinc (Marriott et al. 2008; Koninck, Brand, and Stuckey 2010).

We focus on the refinement-based approach, in which a user writes *abstract* constraint specifications that describe a problem above the level at which modelling decisions are made. Abstract constraint specification languages, such as ESSENCE, and to a lesser extent Zinc, support abstract decision variables with types such as set, multiset, function, and relation, as well as *nested* types, such as set of sets and multiset of functions. Problems can typically be specified very concisely in this way, as demonstrated by the example in Fig. 1. However, existing constraint solvers do not support these abstract decision variables directly, so abstract constraint specifications must be *refined* into concrete constraint models.

```
given      co,ca: int
letting    item be new type of size co
letting    nat be domain int(1..)
given      vol,val: function (total) item → nat
find       x: set of item
maximising ∑ i : x . val(i)
such that  (∑ i : x . vol(i)) ≤ ca
```

Figure 1: The knapsack problem, given in ESSENCE

CONJURE was introduced in prototype form by Frisch *et al.* (2005a). It was able to refine a fragment of ESSENCE limited to nested set and multiset decision variables into models in ESSENCE′, a solver-independent modelling language. Subsequent work (Martínez-Hernández and Frisch 2007; Martínez-Hernández 2008), considered issues involved in automatically channelling among different representations of abstract variables. The subject of this paper is CONJURE 1.0, a major step forward over the previously reported prototypes (henceforth CONJURE 0.X). We summarise its contributions:

- **Coverage of ESSENCE:** CONJURE 1.0 is able to refine all ESSENCE specifications. Whereas CONJURE 0.X demonstrated that refinement could handle types with unbounded nesting, CONJURE 1.0 confirms that refinement can handle all features of ESSENCE.
- **Extensible Domain-specific Rule Language:** The rules used by CONJURE 1.0 are expressed in a new domain-specific declarative language. In contrast each CONJURE 0.X rule was implemented by a piece of Haskell code. This provision separates the tasks of implementation and rule writing and, as we demonstrate, allows the

straightforward addition of new rules to extend the space of possible constraint models CONJURE can produce.

Refinement at a Finer Grain: CONJURE 0.X performed refinement at the level of an individual constraint. This greatly inflates the size of the rule base unnecessarily. Moreover, it requires constraints to be flattened prior to refinement – that is decomposing nested expressions into atomic constraints through the introduction of auxiliary variables As we will show, this process can mask the structure of the original specification and lead to weaker models. By contrast, CONJURE 1.0 rewrites expressions rather than essentially entire constraints, which both reduces the number of rules required and allows us to exploit specification structure to the benefit of the models generated.

Extensive Evaluation: Our working hypothesis is that the kernels[1] of constraint models generated by experts can be automatically generated by refining a problem's specification. The much broader coverage of ESSENCE afforded by CONJURE 1.0 has allowed us to test this hypothesis in much greater detail than previously.

Background

ESSENCE (Frisch et al. 2008) is a language for specifying combinatorial (decision or optimisation) problems. It has a high level of abstraction to allow users to specify problems *without* making constraint modelling decisions, supporting decision variables whose types match the combinatorial objects problems typically ask us to find, such as: sets, multisets, functions, relations and partitions. First introduced by the language is its support for the *nesting* of these types, allowing decision variables of type set of sets, multiset of sets of functions, *etc*. Hence, problems such as the Social Golfers Problem (Harvey 2001), which is naturally conceived of as finding a set of partitions of golfers subject to some constraints, can be specified directly without the need to model the sets or partitions as matrices.

An ESSENCE specification (see (Frisch et al. 2008) for full details), such as that in Fig. 1, identifies: the parameters of the problem class (`given`), whose values are input to specify the instance of the class; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy to be a solution (`such that`). An objective function may also be specified (`min/maximising`) and, for concision, identifiers may be declared (`letting`).

Today's constraint solvers typically support decision variables with atomic types, such as integer or Boolean, have limited support for more complex types like sets or multisets, and no support for nested complex types. Hence, abstract specifications are *refined* by modelling abstract decision variables as constrained collections of variables of unnested primitive types. CONJURE 1.0, like CONJURE 0.X, employs a system of rules to refine ESSENCE specifications into constraint models in ESSENCE′ (Rendl 2010), a language derived from ESSENCE mainly by removing facilities for abstraction and adding facilities common to existing constraint solvers and toolkits. From ESSENCE′ a tool such as TAILOR (Rendl 2010) can be used to translate the model into the format required for a particular constraint solver.

An abstract specification typically can be implemented by many alternative concrete constraint models. CONJURE is intended to generate these alternatives by providing multiple refinement rules for each abstract type, corresponding to the various ways in which a decision variable of that type can be modelled. Furthermore, for each way of modelling the decision variables there can be multiple rules to generate alternative models for a constraint on those variables. Consequently, CONJURE often generates many alternative models for an input specification. We aim to encode each rule that for some problem is used in the generation of some good (or perhaps reasonable) model. Given a problem specification and a set of rules the system generates all possible models. If we have encoded a sufficient set of rules, then the kernels of all good (or reasonable) models of the problem should be contained within the set of models. In future, we will investigate restricting this set to *good* models and the selection of either one recommended model or a portfolio of models with complementary strengths.

The CONJURE 1.0 Architecture

CONJURE 1.0 is structured like a compiler. Its pipeline starts with parsing, validating the input, and type-checking. After these foundation phases, it prepares the input specification for, and performs, refinement, and does some housekeeping:

1. Parsing
2. Validation
   - *Are all identifiers defined?*
   - *Check consistency of declarations. e.g. a function variable cannot be declared both total and partial.*
3. Type Checking
4. Refinement
5. Model Presentation

Phases 1–3 are foundational, while Phase 5 aids perspicuity. Phase 4 is the core of the refinement process, and is the focus of the remainder of the paper. It consists of multiple reentrant levels: following each rule application the process returns to Phase 4i) in case the result of the rule requires the attention of any of the other levels. We summarise Phase 4:

4i) Partial Evaluation CONJURE 1.0 contains a partial evaluator for ESSENCE. This not only simplifies the output models, but also saves the system from applying rules to expressions that can be evaluated readily.

4ii) Representation Selection Refinement of an abstract expression depends crucially on the representation of the abstract decision variables and parameters it involves. Hence, it is natural[2] to select decision variable representations first. This also simplifies the generation of channelling constraints (Level iii) considerably. Typically *structural* constraints are added to the variables in the

[1]By which we mean to exclude advanced features of models, such as symmetry breaking and implied constraints.

[2]Although in contrast with CONJURE 0.X.

concrete representation to ensure that the abstract variable is properly represented. For example, if a set of cardinality *n* is represented by a matrix *M* containing *n* elements then the structural constraint *alldifferent(M)* needs to be added.

4iii) Auto-Channelling Since different occurrences of a variable can have different representations, constraints are necessary to maintain consistency among these different representations. Such constraints are called *channelling* constraints (Cheng et al. 1999). Following previous work (Martínez-Hernández and Frisch 2007; Martínez-Hernández 2008), channelling constraints are generated simply by constraining the different representations of each abstract variable so that they represent the same abstract object. The resultant equality constraints are refined in the same way as any other constraint in the problem specification.

4iv) Expression Refinement Having decided on the representation of each abstract decision variable, it remains to refine the expressions that contain them. Transformations are applied to expressions in a bottom-up manner; namely, smaller expressions constructing larger ones are dealt with earlier in the process. We preserve the ability to apply transformations to larger expressions before their components using rule precedences. Every expression transformation has an associated precedence level and a transformation at a higher precedence level is guaranteed to be applied before those with at relatively lower precedence level by the refinement process.

In order to produce multiple models, refinement branches in two places in Phase 4: Representation Selection and Expression Refinement. Depending on the rules available in the rule base, each abstract decision variable and each expression can be refined in several alternative ways.

A Rule Language for Refinement

To represent the rules at the heart of CONJURE 1.0, we designed a new domain-specific language[3] that provides all and only the facilities we require, rather than use a general system like Cadmium (Duck, Koninck, and Stuckey 2008). CONJURE is run by inputting an ESSENCE specification and a set of rules. This arrangement facilitates the straightforward addition of new rules to extend the space of models CONJURE can produce, as the next section demonstrates.

All CONJURE rules adhere to a single template:

```
<pattern> [⤳ <output>]* [where <guards>]
          [letting <local identifiers>]
```

A rule rewrites an expression that matches against `pattern`, producing one or more `outputs` provided the `guards` are satisfied. A `pattern` is very similar to an ESSENCE domain or expression, but may contain *meta-variables*, identifiers which will be bound to domains or expressions once the pattern matching is performed. To illustrate, the expression $x \cup y = z \cap t$ can be matched against the expression pattern $a = b$, where a is bound to $x \cup y$ and b is bound to $z \cap t$. It can also be matched against $a \cup b = c$, where a is bound to x, b is bound to y and c is bound to $z \cap t$. Both the expressions and patterns use the common operator precedences. Pattern matching on domains, used exclusively by the representation selection rules, operates similarly. The domain $set\ of\ int(x..y + z)$ can be matched against the domain pattern $set\ of\ int(a..b)$, where a is bound to x and b is bound to $y + z$. The same domain can also be matched against the domain pattern $set\ of\ t$, where t is bound to the domain $int(x..y + z)$. Notice, in both ESSENCE and the refinement rule language used by CONJURE, identifiers can refer to both expressions and domains.

Local identifiers are used for concision and to identify new variables created by the refinement process. Based upon this template, the CONJURE rule base contains three types of rule:

Representation Selection Rules label an occurrence of an abstract decision variable in a constraint expression with one of its possible refinements, and add structural constraints to the model associated with that refinement.

Vertical Rules rewrite an expression, *reducing* its level of abstraction. The expression is rewritten with respect to the labels associated with its constituent variables by the representation selection rules.

Horizontal Rules rewrite an expression *without* changing its level of abstraction. These rules are used to reduce the number of vertical rules required. It is important to decrease the number of vertical rules required, since vertical rules need to be given per representation whereas horizontal rules are independent of representation and are only given once and used for all representations.

Thanks to the horizontal rules, expressions involving any ESSENCE operator are transformed to equivalent expressions containing a smaller set of operators. This transformation enables us to use far fewer vertical rules and still be able to refine all valid expressions written in the input language.

The rule language defines a few operators, used mostly for accessing properties of their arguments. The operators used in this paper and their semantics are briefly defined here. Rule language operators are executed at the time of rule application.

`refn` can be used in both vertical rules and representation selection rules.

- In a vertical rule, it accepts an abstract decision variable or a parameter as the argument and returns the refined version.
- In a representation selection rule, it used without any arguments and returns the refined version of the abstract variable or parameter to which the rule is applied.

`repr` can be used in vertical or representation selection rules, accepts an abstract decision variable or a parameter as the argument and returns the name of the representation chosen for its argument.

[3]As noted, CONJURE 0.X's rules were written as fragments of Haskell. Some primitive rules, such as those dealing with partial evaluation, are built into CONJURE 1.0 for efficiency.

indices returns a k-tuple containing the ranges of the index values of a k-dimensional matrix. The usual tuple projection operator (`[]`) can be used to access individual elements of the returned value.

:: is an infix operator to check type constructor matching. Accepts two arguments, an expression and a type constructor name and returns a Boolean value.

domSize accepts a finite domain as an argument, and returns the number of elements in this domain.

We will explain and illustrate the operation of each of the three types of rule by means of an example involving the refinement of the following simple ESSENCE specification:

```
given      x,y : int(0..9)
find       f : function (total) int(0..9) → int(0..9)
such that  f(x) = y, |preimage(f,x)| = y
```

This specification contains two constraints and a single abstract decision variable `f`, a total function `f` mapping digits to digits. `preimage(f,x)` is the set of all values that are mapped to `x` by `f`.

This is a trivial problem, but it serves to illustrate the refinement process. After parsing, validation and type checking, CONJURE arrives at the refinement phase. Representation selection rules are the first to be applied, in Phase 4ii. Consider the following pair of representation selection rules for total functions, which encode one- and two-dimensional matrix representations of a function respectively.

```
function (total) fr → to
    ⇝ Matrix1D
    ⇝ matrix indexed by [fr] of to
    where fr :: int

function (total) fr → to
    ⇝ Matrix2D
    ⇝ matrix indexed by [fr,to] of bool
    ⇝ ∀ i : fr . ((∑ j : to . refn[i,j]) = 1)
    where fr :: int, to :: int
```

These rules output the identifiers for the representation (`Matrix1D` and `Matrix2D`), the variables in the representation and the structural constraints on these variables, if any. The operator `refn` returns the refined version of the abstract variable to which the representation selection rule is applied.

The 1-d matrix is indexed by the domain of the original function whose elements are decision variables with domain equal to the range of the original function. Hence, `f(i)=v` is represented by assigning the `i`th element of the matrix `v`. Since the matrices supported by ESSENCE′ are indexed by integers, there is a guard requiring that the function has an integer domain. There is no such guard on the range of the function — it might have an arbitrarily complex type, in which case the 1-d matrix will be further refined.

The 2-d matrix is indexed by the domain and range of the original function, hence both are required to have an integer domain. Each element of the matrix is a Boolean variable and `f(i) = v` is represented as `m[i,v] = true`. This rule has a third output, a structural constraint that ensures the function is total. This is not necessary in the 1-d representation because each variable must be assigned a value.

In the constraints in the specification, there are two occurrences of `f`. Either of the two rules above may fire on either occurrence, hence refinement branches four ways. We will focus on a single branch in which the 1-d rule has fired on the occurrence in the first constraint, and the 2-d rule has fired on the occurrence in the second. Since the same abstract variable is being represented in two different ways, a channelling constraint is required, and is added in Level iii, giving the intermediate specification:

```
given x,y: int(0..9)
find  f : function (total) int(0..9) → int(0..9)
find  f1: matrix indexed by [int(0..9)] of int(0..9)
find  f2: matrix indexed by [int(0..9),int(0..9)] of bool
such that
  f#Matrix1D(x) = y,
  |preimage(f#Matrix2D,x)| = y,
  ∀ i: int(0..9) . ((∑ j: int(0..9) . f2[i,j]) = 1),
  f#Matrix1D = f#Matrix2D
```

For demonstration purposes, occurrences of `f` in unrefined expressions are labelled using a hash sign with their chosen representation. For instance in the above intermediate problem specification `f#Matrix1D` means the instance of decision variable `f` with representation `Matrix1D`. The vertical rules use the labels to refine these expressions with respect to their chosen representations. To illustrate, consider this vertical rule for function application, where the 1-d matrix representation has been selected for the function:

```
f(i) ⇝ refn(f)[i]
  where f :: function, repr(f) = Matrix1D
```

Here, `refn(f#Matrix1D)` returns the corresponding concrete variable, `f1`. Similarly, `repr(f)` returns the identifier for the representation chosen, corresponding to the first output of the representation selection rules. Following the application of the above rule, the first constraint is rewritten to:

```
    f1[x] = y
```

We now consider an example of a horizontal rule:

```
    |s| ⇝ ∑ i : s . 1 where s :: set
```

This rule replaces the cardinality operator with a summation counting the elements in the set, eliminating the need for a dedicated vertical rule for cardinality. This horizontal rule doesn't result in a poorer model compared to a dedicated vertical rule, it simply decreases the number of vertical rules required. Following its application, the second constraint becomes:

```
    (∑ i : preimage(f#Matrix2D,x) . 1) = y,
```

The following is a vertical rule to refine quantification over inverse function application:

```
    ∑ i : preimage(f,j) . k ⇝ ∑ i : r . m[i,j] * k
        where   f :: function, repr(f) = Matrix2D
        letting m be refn(f), r be indices(m)[1]
```

Note the introduction of local identifiers, `m` and `r`, for concision. This rule operates in the context of the selection of the `Matrix2D` representation, transforming the sum over the elements of the inverse of the function into a summation over the indices of the matrix. Following the application of this rule, the second constraint becomes:

```
    ∑ i : int(0..9) . f2[i,x] = y
```

Partial evaluation has removed the redundant `1`. Finally, we demonstrate the refinement of the channelling constraint between the two representations of the same abstract decision variable. The following horizontal rule removes the need for a vertical rule between every pair or representations for function variables. It transforms an equality between two functions to a universal quantification over the intersection of values being mapped by these two functions and imposing equality on the actual mappings.

```
    a = b ⇝ (defined(a) = defined(b))
              ∧ ∀ i : defined(a) . (a(i) = b(i))
        where a :: function, b :: function
```

When applied to the last constraint `a` is substituted by `f#Matrix1D` and `b` is substituted by `f#Matrix2D`. The first component of the conjunction is then evaluated `true` and subsequently removed from the conjunction. Furthermore, `defined(f#Matrix1D)` is replaced with `int(0..9)` as `f` is a total function. Eventually, the channeling constraint is transformed to the following.

```
    ∀ i : int(0..9) . (f#Matrix1D(i) = f#Matrix2D(i))
```

The vertical rule for handling function application is already given for `Matrix1D`. The following is a similar rule for `Matrix2D`.

```
    f(i) ⇝ ∑ j : r . m[i,j] * j
        where   f :: function, repr(f) = Matrix2D
        letting m be refn(f),  r be indices(m)[1]
```

After the function applications are refined,

```
∀ i: int(0..9). (f1[i] = (∑ j: int(0..9). f2[i,j] * j))
```

This expression is now in normal form, namely, no other rule is applicable to it. The final model generated is:

```
given x,y: int(0..9)
find f1: matrix indexed by [int(0..9)] of int(0..9)
find f2: matrix indexed by [int(0..9),int(0..9)] of bool
such that
  f1[x] = y,
  (∑ i: int(0..9) . f2[i,x]) = y,
  ∀ i: int(0..9) . ((∑ j: int(0..9) . f2[i,j]) = 1),
  ∀ i: int(0..9). f1[i] = (∑ j: int(0..9). f2[i,j] * j)
```

CONJURE automatically removes the `find f` statement when `f` no longer occurs in any of the constraints.

The output of CONJURE is ESSENCE′ annotated with some *metadata*. The metadata enables any solution to an instance of an ESSENCE′ model to be automatically re-expressed as a solution to the original ESSENCE specification. The metadata are inserted to the output model file as line comments and tools dealing with ESSENCE′ input are simply expected to ignore them.

The CONJURE Rule Base and Its Extension

Refining ESSENCE specifications is a complex task. In order to cover the entire ESSENCE language, CONJURE must be able to cope with arbitrarily nested expressions of decision variables with arbitrarily nested abstract types. In order to deal with this complexity, CONJURE 0.X resorted to *flattening* an input specification: decomposing nested expressions into atomic constraints through the introduction of auxiliary variables. A refinement rule was then provided for each flat constraint (for the fragment of ESSENCE that CONJURE 0.X considered). However, the flattening process can mask the structure of the original specification and lead to weaker models as the following example demonstrates.

Consider the constraint: `(a ∪ b) ⊆ (c ∩ d)`. Flattening this constraint gives the constraints `aux0 = a ∪ b`, `aux1 = c ∩ d`, `aux0 ⊆ aux1` containing two auxiliary set variables. This is expensive: each auxiliary set variable incurs the cost of a set of concrete variables and constraints to represent it. This cost increases considerably if the sets have nested types. If we do not flatten, we can use horizontal rules to rewrite the expression to a far more efficient pair of quantified expressions:

```
∀ i : a . (i ∈ c ∧ i ∈ d),  ∀ i : b . (i ∈ c ∧ i ∈ d)
```

As we have seen, therefore, CONJURE 1.0 refines at a *finer grain* than CONJURE 0.X, refining expressions rather than entire constraints, hence eliminating the need for flattening. Even so, a large number of rules are ostensibly required to refine all of ESSENCE. The key to reducing this number are the horizontal rules, which rewrite expressions into a form for which we have a vertical rule, as per the cardinality horizontal rule in the previous section. Horizontal rules are *representation independent* — a horizontal rewrite of an expression is valid irrespective of the concrete representation of its constituent variables. Hence, the horizontal rules reduce the set of vertical rules required and the set of horizontal rules does not grow as new representations are added.

Using horizontal rules we have been able to reduce the set of rules required for each ESSENCE type. For each representation of an abstract type, a representation selection rule is required as well as a small set of vertical rules. For set, multiset, relation, and partition variables, one vertical rule per quantifier is necessary. For functions, vertical rules are required for function application, quantification over the inverse of the function and quantification over the 'defined' elements of a partial function. Consequently, at present CONJURE 1.0 has 37 representation selection rules, and 96 vertical and horizontal rules.

The use of horizontal rules to keep vertical rules to a minimum also reduces the effort of adding new representations to the system. To illustrate, consider adding rules to support the Gent representation of sets (Jefferson 2007). The properties of this representation are irrelevant — this example is chosen as being a non-trivial, unusual representation of a set typical of a representation one might wish to add to those already present in the rule base. In brief, a set S of integers drawn from some range d is represented by a matrix g, indexed by d, of decision variables with domain $0..n$, where n is either the fixed or the maximum cardinality of S. Element i is in S iff $g[i]$ is non-zero. Furthermore, the non-zero elements of g are required to be in ascending order. If, for example, $S = \{1, 2, 4\}$, drawn from $1..5$, then $g = [1, 2, 0, 3, 0]$. Just four rules, in addition to the existing horizontal rules, are necessary to integrate the

```
set of τ                              [Representation selection]
  ⇝ Gent
  ⇝ matrix indexed by [τ] of int(0..n)
  ⇝ ∀ i : τ . ((refn[i] = 0) ∨
      (refn[i] = 1 +
        (∑ j : int(..i-1) ∩ τ . (refn[j] > 0))))
  where   τ :: int
  letting n be domSize(τ)
```

```
∀ i : s . k                                        [Vertical]
  ⇝ ∀ i : r . (m[i] > 0) ⇒ k
  where   s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

```
∃ i : s . k                                        [Vertical]
  ⇝ ∃ i : r . (m[i] > 0) ∧ k
  where   s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

```
∑ i : s . k                                        [Vertical]
  ⇝ ∑ i : r . (m[i] > 0) * k
  where   s :: set, repr(s) = Gent
  letting m be refn(s), r be indices(m)[0]
```

Figure 2: The four refinement rules sufficient to integrate the Gent representation of sets into CONJURE.

Gent representation fully into CONJURE 1.0 (Fig.2), such that any of the set operators available in ESSENCE can be refined. In this respect, CONJURE 1.0 is clearly superior to Conjure 0.X, which would require a distinct rule for every operator and for every combination of representations of the arguments of that operator. We estimate that CONJURE 0.X would require up to 85 rules to add the Gent representation in addition to the existing two set representations. Adding a fourth representation to CONJURE 0.X would require even more rules.

Evaluation

A milestone achieved uniqely by CONJURE 1.0 is full coverage of ESSENCE: it has at least one variable representation rule for every abstract variable type, and rules for the operators defined on them. We now test the hypothesis that the kernels of constraint models written by experts can be automatically generated by refining a problem's specification. We wrote specifications for a diverse set of 32 benchmark problems drawn from the literature and refined them with CONJURE. Table 1 presents the results: the number of generated models, papers that contain a kernel CONJURE 1.0 generate and the abstract parameters and variables involved in the problem. Papers containing n kernels generated by CONJURE 1.0 are labelled $\times n$. Notice the variety of decision variable types involved in the benchmark problems, representing a proof that the current collection of rules, the rewrite rule language, and the Conjure system as a whole is capable of refining a variety of abstract problem specifications into concrete models.

The number of models generated for a problem specification depends on the number of representation options for the involved abstract decision variables. For instance, the *Maximum Density Still Life* contains a set decision variable whose elements are tuples and currently the system has only one variable selection rule that matches this type. Problems such as *Magic Hexagon* only contain decision variables that are concrete, so do not require refinement. We did find papers containing kernels which we are currently unable to generate, for example for *Langford's Number Problem* and *Maximum Density Still Life*. These come from complex reformulations of the problem. In each of these cases, an alternative ESSENCE specification would allow CONJURE to generate the missing kernel.

Further research is necessary to improve the quality of generated models. This is not surprising since producing a good model is well known to be difficult even by human modellers. We have established that good rewrite rules are applicable to many problems and we hope as our refinement rules database improves further, we will produce better models for all problems.

The following is an example where a specific rule helps producing better models. A common method of iterating over pairs of distinct elements in a set is to use an expression like $\forall$i, j : s.(i $\neq$ j) $\Rightarrow$ k. Constraints of this form arise in many problems, including *Golomb ruler* and *Social golfers*. Our standard rules would refine i $\neq$ j using the representation chosen for the elements of s, which can lead to a complex constraint. However, if s is represented with the explicit representation then i $\neq$ j is true if and only if i and j are represented by different elements of the matrix in the refinement. This leads to the rule:

```
∀ i,j : s . (i ≠ j) ⇒ k ⇝
  ∀ i',j' : r . (i' ≠ j') ⇒ k {i ↦ m[i'],j ↦ m[j']}
  where   s :: set, repr(s) = Explicit
  letting m be refn(s), r be indices(m)[0]
```

The substitution operator used in the above rule is a part of the rule language: a {b ↦ c} replaces every occurrence of b found in a with c.

Conclusion and Future Work

We have presented the CONJURE 1.0 automated constraint modelling system and demonstrated its ability to reproduce the kernels of the constraint models of 32 benchmark problems found in the literature. It achieves full coverage of the ESSENCE language via a new domain-specific rule language, whose features include: fine-grained refinement to avoid the need for flattening, which, as we have demonstrated, can impair the models produced; horizontal rules that normalise expressions to reduce considerably the total number of rules necessary for refinement; easy extensibility.

In future we of course wish to go beyond model kernels to produce full models of the same quality as those found in the literature, including symmetry breaking and implied constraints. CONJURE's flexible rule-based architecture is ideally placed to achieve these aims in large part by adding new rules to those available (cf. the example in the previous section). Furthermore, we will prune the set of models produced to contain only the most effective models. In part, we plan to achieve this by applying a prioritisation system to rule application. This will allow refinement paths that are provably superior to dominate those shown to be weaker.

Table 1: Runnning CONJURE on benchmark problems.

Problem name	Models	Reference	Nb. abstract params[2] and vars
Car Sequencing	128	(Gravel, Gagné, and Price 2005)	*4 functions, 1 relation*
Template Design	16	(Proll and Smith 1998)	*2 function variables, 1 mapping msets to integers*
Low Autocorellation Binary Sequences	4	(Gent and Smith 1999)	*1 function*
Golomb Ruler	81	(Smith, Stergiou, and Walsh 2000; Prestwich 2003)	*1 set*
All-interval series	8	(Choi and Lee 2002)	*2 functions*
Vessel loading	256	(Brown 1998)	*9 functions, 1 mapping from a set*
Perfect Square Placement	1024	(Cambazard and O'Sullivan 2010)	*2 functions*
Social Golfers	3	(Kiziltan and Hnich 2001; Hawkins, Lagoon, and Stuckey 2005)	*multiset of partitions*
Progressive Party	81	(Smith et al. 1995)	*1 set, 1 set of functions*
Schur's Lemma	81	(Flener et al. 2002b)$\times 2$	*1 partition*
Traffic Lights	2	(Hower 1998)	*1 set of functions mapping integers to tuples*
Magic Squares	1	(Refalo 2004)	*1 2-dimensional matrix*
Bus Driver Scheduling	27	(Muller 1998)	*1 set of sets, 1 partition*
Magic Hexagon	1	Model from CSPLib 23	*1 2-dimensional matrix*
Langford's Number Problem	32	(Hnich, Smith, and Walsh 2004)	*1 function*
Round Robin Tournament Scheduling	27	(Frisch, Jefferson, and Miguel 2004)	*1 relation between 2 integers and 1 set*
BIBD	16	(Petrie 2005)	*1 relation between 2 unnamed types*
Balanced Academic Curriculum Problem	512	(Hnich, Kiziltan, and Walsh 2002)	*2 functions, 1 relations*
Rack Configuration Problem	288	(Kiziltan and Hnich 2001)	*7 functions, 1 mapping integers to sets*
Maximum Density Still Life	1	(Smith 2006)	*1 set of tuples*
Word Design for DNA Computing	16	Model from CSPLib 33	*1 set of functions*
Warehouse Location Problem	16	(Van Hentenryck 1999)	*3 functions, 1 mapping tuples to integers*
Fixed Length Error Correcting Codes	16	(Frisch, Jefferson, and Miguel 2003)	*2 functions, 1 mapping tuples to integers*
Steel Mill	4	(Flener et al. 2002a)	*3 functions, 1 from sets*
N-Fractions Puzzle	16	(Frisch, Jefferson, and Miguel 2004)	*1 function*
Steiner Triple Systems	9	(Kiziltan and Hnich 2001; Hawkins, Lagoon, and Stuckey 2005)	*1 set of sets*
N-Queens Problem	4	(Hnich, Smith, and Walsh 2004)$\times 2$	*1 function*
Peaceably Co-existing Armies of Queens	1	(Smith, Petrie, and Gent 2004)	*1 set of tuples*
Maximum Clique Problem	81	(Regin 2003)	*1 set, 1 set of sets*
Graph Colouring	4	(Hao and Dorne 1996; Chang, Chen, and King 1997)	*1 function*
SONET Configuration	27	(Frisch et al. 2005b)[1]	*1 mset of sets, 1 set of sets*
Knapsack Problem	36	(Sellmann 2009)	*2 functions, 1 set*

[1] Some models in this paper have set variables, which CONJURE currently always refines.

[2] Since CONJURE operates at the problem class level, problem parameters need to be refined as well as decision variables.

Acknowledgements

Ozgur Akgun is supported be a Scottish Informatics and Computer Science Alliance (SICSA) scholarship. This research is supported by UK EPSRC grant no EP/H004092/1.

References

Bessiere, C.; Coletta, R.; Koriche, F.; and Sullivan, B. O. 2006. Acquiring constraint networks using a SAT-based version space algorithm. In *AAAI 2006*, 1565–1568.

Brown, K. N. 1998. Loading supply vessels by forward checking and unenforced guillotine cuts. In *17th Workshop of the UK Planning and Scheduling SIG*.

Cambazard, H., and O'Sullivan, B. 2010. Propagating the Bin Packing Constraint Using Linear Programming. In *CP 2010*. 129–136.

Chang, C.-M.; Chen, C.-M.; and King, C.-T. 1997. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications* 34(9):1 – 14.

Charnley, J.; Colton, S.; and Miguel, I. 2006. Automatic generation of implied constraints. In *Proc. of ECAI 2006*, 73–77. IOS Press.

Cheng, B.; Choi, K. M. F.; Lee, J. H. M.; and Wu, J. C. K. 1999. Increasing constraint propagation by redundant modeling: an experience report. *Constraints* 4(2):167–192.

Choi, C., and Lee, J. 2002. On the pruning behaviour of minimal combined models for permutation csps. In *CP 2002 Workshop on Reformulation*.

Duck, G. J.; Koninck, L. D.; and Stuckey, P. J. 2008. Cadmium: An implementation of acd term rewriting. In *ICLP*, 531–545.

Flener, P.; Frisch, A. M.; Hnich, B.; Kiziltan, Z.; Miguel, I.; and Walsh, T. 2002a. Matrix modelling: Exploiting common patterns in constraint programming. In *the International Workshop on Reformulating CSPs*, 27–41.

Flener, P.; Frisch, A. M.; Hnich, B.; Kiziltan, Z.; Miguel, I.; Pearson, J.; and Walsh, T. 2002b. Breaking row and column symmetries in matrix models. In *CP 2002*, 462–476.

Flener, P.; Pearson, J.; and Ågren, M. 2003. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR 2003*, 214–232.

Frisch, A. M.; Jefferson, C.; Hernandez, B. M.; and Miguel, I. 2005a. The rules of constraint modelling. In *Proc. of the IJCAI 2005*, 109–116.

Frisch, A. M.; Hnich, B.; Miguel, I.; Smith, B. M.; and Walsh, T. 2005b. Transforming and refining abstract constraint specifications. In *6th Symposium on Abstraction, Reformulation and Approximation*, 76–91. Springer.

Frisch, A. M.; Harvey, W.; Jefferson, C.; Martínez-Hernández, B.; and Miguel, I. 2008. Essence: A constraint language for specifying combinatorial problems. *Constraints 13(3)* 268–306.

Frisch, A. M.; Jefferson, C.; and Miguel, I. 2003. Constraints for breaking more row and column symmetries. In *CP 2003*, 318–332.

Frisch, A. M.; Jefferson, C.; and Miguel, I. 2004. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *ECAI 2004*, 171–175.

Gent, I. P., and Smith, B. 1999. Symmetry breaking during search in constraint programming. In *ECAI'2000*, 599–603.

Gravel, M.; Gagné, C.; and Price, W. L. 2005. Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem. *Journal of the Operational Research Society* 56(11):1287–1295.

Hao, J.-K., and Dorne, R. 1996. Empirical studies of heuristic local search for constraint solving. In *CP '96*. 194–208.

Harvey, W. 2001. Symmetry breaking and the social golfer problem. In *Proceedings SymCon-01: Symmetry in Constraints, co-located with CP 2001*, 9–16.

Hawkins, P.; Lagoon, V.; and Stuckey, P. J. 2005. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR)* 24:109–156.

Hnich, B.; Kiziltan, Z.; and Walsh, T. 2002. Modelling a balanced academic curriculum problem. In *CP-AI-OR-2002*, 121–131.

Hnich, B.; Smith, B. M.; and Walsh, T. 2004. Dual modelling of permutation and injection problems. *J. Artif. Int. Res. (JAIR)* 21:357–391.

Hnich, B. 2003. Thesis: Function variables for constraint programming. *AI Commun* 16(2):131–132.

Hower, W. 1998. Revisiting global constraint satisfaction. *Information Processing Letters* 66(1):41–48.

Jefferson, C. 2007. *Thesis: Representations in Constraint Programming*. Ph.D. Dissertation, University of York.

Kiziltan, Z., and Hnich, B. 2001. Symmetry breaking in a rack configuration problem. In *the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints*.

Koninck, L. D.; Brand, S.; and Stuckey, P. J. 2010. Data independent type reduction for zinc. In *ModRef10*.

Little, J.; Gebruers, C.; Bridge, D. G.; and Freuder, E. C. 2003. Using case-based reasoning to write constraint programs. In *CP*, 983.

Marriott, K.; Nethercote, N.; Rafeh, R.; Stuckey, P. J.; de la Banda, M. G.; and Wallace, M. 2008. The design of the zinc modelling language. *Constraints 13(3)*.

Martínez-Hernández, B., and Frisch, A. M. 2007. The automatic generation of redundant representations and channelling constraints. In *Trends in Constraint Programming*. ISTE. chapter 8, 163–182.

Martínez-Hernández, B. 2008. *Thesis: The Systematic Generation of Channelled Models in Constraint Satisfaction*. Ph.D. Dissertation, University of York.

Mills, P.; Tsang, E.; Williams, R.; Ford, J.; and Borrett, J. 1999. EaCL 1.5: An easy abstract constraint optimisation programming language. Technical report, University of Essex, Colchester, UK.

Muller, T. 1998. Solving set partitioning problems with constraint programming. In *PAPPACT98*.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack., G. 2007. Minizinc: Towards a standard CP modelling language. In *Proc. of CP 2007*, 529–543.

Petrie, K. 2005. *Constraint Programming, Search and Symmetry*. Ph.D. Dissertation, University of Huddersfield.

Prestwich, S. 2003. Negative effects of modeling techniques on search performance. *Annals of Operations Research* 118:137–150. 10.1023/A:1021809724362.

Proll, L., and Smith, B. 1998. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS J. on Computing* 10:265–275.

Puget, J.-F. 2004. Constraint programming next challenge: Simplicity of use. In *Principles and Practice of Constraint Programming - CP 2004*, 5–8.

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *CP 2004*, volume 3258. 557–571.

Regin, J.-C. 2003. Using constraint programming to solve the maximum clique problem. In *CP 2003*, 634–648.

Rendl, A. 2010. *Thesis: Effective Compilation of Constraint Models*. Ph.D. Dissertation, University of St. Andrews.

Sellmann, M. 2009. Approximated consistency for the automatic recording constraint. *Comput. Oper. Res.* 36:2341–2347.

Smith, B. M.; Brailsford, S. C.; Hubbard, P. M.; and Williams, H. P. 1995. The progressive party problem: Integer linear programming and constraint programming compared. In *CP '95*, 36–52.

Smith, B. M.; Petrie, K. E.; and Gent, I. P. 2004. Models and symmetry breaking for peaceable armies of queens. In *Integration of AI and OR Techniques in CP for COP*. 271–286.

Smith, B. M.; Stergiou, K.; and Walsh, T. 2000. Using auxiliary variables and implied constraints to model non-binary problems. In *17th National Conference on AI*, 182–187. AAAI Press.

Smith, B. 2006. A dual graph translation of a problem in life. In *CP 2002*, volume 2470. Springer Berlin / Heidelberg. 89–94.

Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*. Cambridge, MA, USA: MIT Press.