The Rules of Modelling:

Towards Automatic Generation of Constraint Programs

Alan M. Frisch, Chris Jefferson,
Bernadette Martínez Hernández, and Ian Miguel

AI Group, Dept. Computer Science, University of York, UK
{frisch,caj,berna,ianm}@cs.york.ac.uk

Abstract. Many and diverse search problems have been solved with great success using constraint programming. However, to apply constraint programming to a particular domain, the problem first must be *modelled* as a constraint satisfaction or optimisation problem. Since constraints provide a rich language, typically many alternative models exist. Formulating a *good* model therefore requires a great deal of expertise. This paper describes Conjure, a system that *refines* an abstract specification of a problem into a set of alternative constraint programs. Refinement is *compositional*: alternative constraint programs are generated by composing refinements of the components of the specification. Since the specification language (Essence) is significantly closer than a constraint program to the way in which problems are commonly specified in natural language, the modelling bottleneck is substantially reduced.

1 Introduction

To employ constraint programming technology to solve a problem, the problem first must be characterised, or *modelled*, by a set of constraints that solutions must satisfy. Modelling can be difficult and requires expertise, thus limiting widespread use. The vast majority of research on modelling with constraints presents alternative models to a particular problem and evaluates these alternatives through analysis and/or experiment. The process by which the alternative models are generated is rarely, if ever, discussed. Each constraint programmer must learn the art of modelling by forming generalisations from these studies.

We show that a set of rules can formalise the generation of alternative models. We need a language in which to express the abstract problem structure of which models are a function. If all modelling choices are to be open to the rules, the language must have a level of abstraction above that at which modelling decisions are made. We have designed such a language and call it Essence (Section 3).

The rules are embedded in the Conjure system, which generates models of the type supported by existing constraint programming languages from an Essence specification. Hence, we refer to our rules as *refinement rules*. The intention is to generate models including those that a human expert would generate. Currently, Conjure may also generate correct but inefficient models.

A proper formalisation of modelling must account for the major functions performed by a human expert. Our rules account for four such functions:

Representing complex decision variables: Currently, constraint programming provides decision variables whose domains or domain values are finite sets of atomic elements. Yet, combinatorial problems often require finding a non-atomic combinatorial structure. For example, the Social Golfers Problem (Problem 10 at www.csplib.org) requires partitioning a set G of golfers in each week of play. Thus the goal is to find a set of partitions of G—that is, a set of sets of sets of G. Modelling the Social Golfers Problem requires deciding how to represent the decision variable, whose type is set of sets of sets of G, as a structured collection of atomic decision variables. Representing complex decision variables is the central task in modelling many problems. Section 5 shows how the alternatives can be formalised in a set of rules.

Exploiting channelling: There are often many ways to represent a complex decision variable and expert modellers know that it is sometimes advantageous to use multiple representations simultaneously and impose channelling constraints to keep the representations consistent with each other. Section 7 shows how such models can be generated by formal rules.

Identifying and breaking symmetry: Many models contain symmetries, often enormous numbers of them. Symmetries in the model result in redundancies in the search space. Expert modellers are able to identify symmetries in a model and break them, either by introducing symmetry-breaking constraints or using a symmetry-aware search method. It has been argued that detecting symmetries in a model is as hard as graph isomorphism. We maintain that symmetries enter a model from two sources: either a symmetry is inherent to the combinatorial problem or it is introduced by the modelling process. In Section 4.1 we show how our model-generating rules can be equipped to report the symmetries that they introduce into the model. This obviates the expensive procedure of detecting these symmetries in the resulting model.

Performing transformations: A human modeller can improve a model by performing transformations such as introducing implied constraints or auxiliary variables. Both abstract specifications and concrete models can be transformed. Transformations, unlike refinement, do not change the level of abstraction in a specification or model. Transformations have been the topic of work reported elsewhere [8], so are not discussed herein.

2 Challenges and Opportunities

The key challenge in formulating a system of refinement rules is to make it *compositional*—that is, the refinement of an expression is composed from the refinements of its constituents. A compositional system is the only reasonable way to deal with a language that allows arbitrary (unbounded) nesting. Our specification language *must* admit arbitrary nesting of two kinds. First, as with all constraint languages, an expression (e.g. a constraint) can be composed of arbitrarily complex subexpressions. Second, and uniquely, the type of a decision variable can be nested arbitrarily deep. For example, the language allows for decision variables to be of type set, set of sets, set of sets of sets, and so forth.

Formulating compositional refinement rules is difficult. Unlike standard compilers, decision variables of a non-atomic type usually can be refined in multiple ways. However, not all refinements support all operations: context restricts which are acceptable. Refinements of an operator may differ depending on the refinements of its operands. Hnich [11] tackles these issues for variables only of type function over non-nested elements. His system, Fiona, unlike Conjure, has only a limited ability to produce models with multiple representations and channelling. Handling nested types, such as sets of sets, is much more difficult; *full* compositionality is needed to handle unbounded nesting. The acceptable representations of the outer type must be determined without looking arbitrarily deep into the nesting.

Applying operators to nested types provides the biggest challenge. Suppose that A and B are sets of some deeply nested type and we wish to refine the constraint $A = B$. Such a constraint would have to involve all components of both A and B; and since the type of A and B could be nested to arbitrary depth, this complex constraint would have to be generated through recursive rule applications. Furthermore, Conjure can produce refinements in which A and B do not have the same kind of representation.

Developing a formal account of model generation is difficult but worthwhile. A rigorous account of model generation can make our study of modelling more systematic, revealing gaps in our understanding. It could also guide the study of model selection by identifying the decision points and the set of alternatives available at each. Furthermore, it would be useful in teaching and presenting modelling and could be used to construct a catalogue of modelling constructs. Our rules are used by Conjure to generate all possible models, but the rules could also be used within a system like an interactive theorem prover. At a decision point the system could present the alternatives and request a choice from the user. A modeller using the system could only produce correct models. Finally, automated model generation is a major step towards automating the entire modelling process.

3 An Introduction to Essence

This section introduces the abstract specification language Essence. We begin by considering the Golomb Ruler Problem (GRP, problem 6 at www.csplib.org. See Figure 1). A specification is a list of statements, of which there are six kinds,

Given n, put n integer ticks on a ruler of size m such that all inter-tick distances are unique. Minimise m.

`given`	n: `nat`
`letting`	$bound$ `be` 2^n
`find`	$Ticks$ `set (size` n`) of` $0..bound$
`minimising`	$\max(Ticks)$
`such that`	$\forall \{i,j\} \subseteq Ticks. \forall \{k,l\} \subseteq Ticks. \{i,j\} \neq \{k,l\} \rightarrow \lvert i-j \rvert \neq \lvert k-l \rvert$

Fig. 1. Specification of the Golomb Ruler problem.

signalled by the keywords `given`, `letting`, `find`, `maximising`, `minimising` and `such that`. Statements are composed into specifications according to:

(`given` | `letting`)* `find`+ [`minimising` | `maximising`] `such that`*

Identifiers in CONJURE come in four categories: constant, parameter, quantified variable, and decision variable. "Letting" statements declare constant symbols and give their values. "Given" statements declare the problem's parameters; the values of the parameters are provided to specify the instance of the problem class. Parameter values are not part of the problem specification; as in OPL and ESRA, they are provided elsewhere. "Find" statements declare decision variables. A "minimising" or "maximising" statement gives the objective function, if any. Finally, "such that" statements give the problem's constraints.

The GRP specification begins by declaring n to be a parameter and *bound* to be a constant. Since n is used in the declaration of *bound*, the declaration of *bound* must come after the declaration of n. Every symbol must be defined before it is used. This restriction prevents cyclical definitions and means that decision variables cannot be used in the definitions of constants and parameters.

The specification language is strongly typed and every expression has a type. The type of the decision variable T indicates that the goal of the problem is to find a set containing n elements, each of which is an integer in the range 0 to *bound*. The types supported by ESSENCE include the usual atomic types such as `int` (integer), `nat` (natural), `bool` (Boolean) and $\alpha..\alpha'$ (a range type), where α and α' are integer expressions. ESSENCE also provides enumerated types and a new and very useful atomic type: `type` (`size` α), a type of α unnamed elements.

ESSENCE is the first constraint language to support fully-compositional type constructors. So, for example, a decision variable may be of type integer, of type set of integer, of type set of set of integer, and so forth. More formally, if $\tau, \tau_1, \ldots, \tau_n$ are types and α, and α' are expressions of type natural and $I_1, \ldots, I_n$ are of type range then the following are among the types of ESSENCE:

$\tau_1 \times \cdots \times \tau_n$	cross product
`set (size` α`) of` τ, `set (maxsize` α`) of` τ	two kinds of set
`mset (size` α`) of` τ, `mset(maxsize` α`) of` τ	two kinds of multiset
`matrix (indexed by` $I_1, \ldots, I_n$`) of` τ	matrix
`partition of` τ, `partition (size` α`) of` τ	two kinds of partition
`rpartition (size` α`) of` τ	regular partition
$\tau_1 \rightarrow \tau_2$	partial function from τ_1 to τ_2

Constraints in this specification language are built from the parameters, constants and decision variables using operators commonly found in mathematics and in other constraint specification languages. The language also includes variable binders such as $\forall x$, $\exists x$ and $\sum_x$, where x can range over any specified finite type. Each subexpression in a constraint is typed. The constraint in the GRP can be paraphrased in English as "For any unordered pair $\{i, j\}$ of ticks and any unordered pair $\{k, l\}$ of ticks, if the two pairs are different then the distance between i and j is not the same as the distance between k and l." To clarify the notation, the expression $\{i, j\} \subseteq T$ means that two distinct elements are drawn from T and, without loss of generality, call one of them i and the other j.

We also consider the Sonet problem, (Figure 2). Notice that *Nodes* is declared to be a range. Since the range is parameterised by *nnodes*, the declaration of *nnodes* must come first. And since the definition of *demand* uses *Nodes*, it must

A Sonet communication network comprises a number of rings, each joining a number of nodes. A node is installed on a ring using an ADM and there is a capacity bound on the number of nodes that can be installed on a ring. Each node can be installed on more than one ring. Communication can be routed between a pair of nodes only if both are installed on a common ring. Given the capacity bound and a specification of which pairs of nodes must communicate, allocate a set of nodes to each ring so that the given communication demands are met. The objective is to minimise the number of ADMs used. (This is a common simplification of the full Sonet problem, as described in [7])

`given`	*nrings*:`nat`, *nnodes*:`nat`, *capacity*:`nat`
`letting`	*Nodes* `be` 1..*nnodes*
`given`	*demand*:`set (size` m`) of set (size 2) of` *Nodes*
`find`	*rings*: `mset (size` *nrings*`) of set (maxsize` *capacity*`) of` *Nodes*
`minimising`	$\sum_{r \in rings} \lvert r \rvert$
`such that`	$\forall pair \in demand.\ \exists r \in rings\ .\ pair \subseteq r$

Fig. 2. Specification of the Sonet problem.

come last. A subtle point is that the third line of the specification is declaring *two* parameters. When the *demand* parameter is instantiated to a particular set of sets, the size of the outer set will be known. Hence, the value of m is given indirectly. This declaration also requires the inner sets to have cardinality two. The goal is to find a multiset (representing the rings), each element of which is a set of *Nodes* (representing the nodes on that ring). The constraint ensures that any pair of nodes that must communicate are installed on a common ring.

3.1 Target Language

Essence specifications are refined into a target language called Essence′, which is a subset of Essence similar to existing constraint languages, OPL being the closest. The only types Essence′ has are integers and matrices (of any dimension) indexed by integers. It does not have enumerated types or types of unnamed elements. Binders, such as quantifiers and summations, range only over finite integer intervals. From this generic constraint language, it is a short step to an established constraint language, such as OPL itself, Solver, or Eclipse. To perform this step, we are developing a suite of back-end translators. Note that, to refine to a language that supports variables whose domains are *sets* of atomic values we need only omit the refinement of such sets. In future, we also intend to refine to more restricted languages, such as SAT and Pseudo-Boolean formulations.

4 The Architecture of Conjure

This section describes Conjure, a system for automatically generating a set of Essence′ models from Essence specifications. At the core of Conjure are a set of *refinement rules*, which refine an Essence expression into a set of Essence′ expressions. For reasons of space we give only a subset of Conjure's refinement rules. The remainder, some of which are very straightforward, follow a similar pattern. We begin by describing the refinement of a single Essence expression, before discussing how a whole specification is refined.

4.1 Refining an ESSENCE Expression

An important consideration in refining the Sonet specification (Figure 2) is the representation of each *ring* (a set of *Nodes* with maximum size *capacity*). We will consider two alternatives, the *occurrence* (characteristic function) representation and the *variable-sized explicit* representation. The occurrence representation is a one-dimensional matrix indexed by *Nodes* with Boolean elements, where a 'true' entry indicates that a particular node is installed on the ring. To preserve the fact that the set contains at most *capacity* elements, the sum of the matrix is constraind to be at most *capacity*; we assume that summation of Booleans involves treating 'true' as 1 and 'false' as 0. The variable-sized explicit representation is a pair of one-dimensional matrices of size *capacity*. The first matrix records each of the (up to) *capacity* elements of the set explicitly. The second contains Boolean 'switches' that indicate whether the corresponding element is actually in the set. Again, to ensure that the set is properly represented, the elements of the first matrix are constrained to take distinct values when assigned.

The refinement described above, i.e. of a set of at most n elements of type τ, is the responsibility of the BOUNDEDSET rule (Figure 3). This rule demonstrates several important features of our refinement rules. We consider these in turn, but first note the general form of the rule. Refinement is performed via the ρ operator, which, since there are often many representations of an abstract object, maps an ESSENCE expression to a *set* of alternative ESSENCE′ expressions. ρ is defined by a set of uniquely-named equalities of the form $R\ \rho(e) = e'_1 \cup e'_2 \cup \cdots \cup e'_n$, where R is the name of the equation, e is an ESSENCE expression and each e'_i is a set of ESSENCE′ expressions, usually given via a set comprehension. For ease of presentation, each rule is split into clauses, written: $R_1\ \rho(e) \stackrel{\text{ref}}{\rightarrow} e'_1$, $R_2\ \rho(e) \stackrel{\text{ref}}{\rightarrow} e'_2$, …, $R_n\ \rho(e) \stackrel{\text{ref}}{\rightarrow} e'_n$. In the case of BOUNDEDSET, the first and second clauses perform the refinement to the occurrence and variable-sized explicit representations respectively. To illustrate the operation of a rule, we now discuss in detail how BOUNDEDSET refines *ring*.

BOUNDEDSET1 refines a set S into a matrix denoted by S' while BOUNDEDSET2 refines S into a pair of matrices denoted by S' and *Switches*. When a rule needs to introduce a new identifier, such as those denoted by S' and *Switches*, it does so by making use of the `genSymbol`() function. This function takes two arguments. The first is either an explicit category (see Section 3), or an identifier from which category information can be extracted. The second is the type of the new identifier, such as a Boolean matrix used for the occurrence representation. The `genSymbol`() function creates a unique identifier of the required category and type. When, for example, refining *ring* with BOUNDEDSET1, $S' =$ `genSymbol`(*ring*, `matrix (indexed by` *Nodes*`) of bool`), and so S' will denote a unique identifier for a matrix of decision variables (since *ring* is a decision variable). Definitions via `genSymbol`() and type information[1] (except in the case of the input expression) are given on the right of the long vertical bar. The bar

[1] '_' denotes 'don't care'

BoundedSet1 $\rho(S$:set (maxsize n) of $\tau) \overset{\text{ref}}{\rightarrow}$

$$
\begin{array}{l|l}
\{\ S'' & S' = \texttt{genSymbol}(S, \texttt{matrix (indexed by } \tau\texttt{) of bool}) \\
\quad \texttt{such that } \phi & S'' \texttt{:matrix (indexed by \_) of bool} \\
\quad \texttt{represent } S \texttt{ by } occset(S') & \\
\quad | & \\
\quad \phi \in \rho(\text{sum}(S') \leq n) & \\
\quad S'' \in \rho(S') & \\
\} &
\end{array}
$$

BoundedSet2 $\rho(S$:set (maxsize n) of $\tau) \overset{\text{ref}}{\rightarrow}$

$$
\begin{array}{ll|l}
\{\ \langle S'', Switches\rangle & & S' = \texttt{genSymbol}(S, \texttt{matrix (indexed by } 1..n\texttt{) of } \tau) \\
\quad \texttt{such that} & \phi & Switches = \\
\quad \texttt{represent} & S \texttt{ by } varexpset(\langle S', Switches\rangle) & \quad \texttt{genSymbol}(S, \texttt{matrix (indexed by } 1..n\texttt{) of bool}) \\
\quad \texttt{symmetry} & indexsym(\langle S', Switches\rangle, 1) & S'' \texttt{:matrix (indexed by } 1..n\texttt{) of \_} \\
\quad | & & \\
\quad \phi \in \rho(\text{AllDifferent}(S')) & & \\
\quad S'' \in \rho(S') & & \\
\} & &
\end{array}
$$

Fig. 3. The BoundedSet refinement rule.

itself has no meaning beyond separating the details of the clause from the types and definitions.

In our example, the Boolean matrix indexed by the integer range $1..nnodes$ generated by BoundedSet1 is in Essence′ and no further refinement is necessary. However, this cannot be assumed to be the case in general. Since Essence supports arbitrary type nesting, τ, the type of the elements of S, may be some nested type. Therefore, ρ is applied recursively to S' in both clauses and it is the result of this refinement, denoted by S'', that is the final result of applying BoundedSet1. BoundedSet2 operates similarly, with the exception that *Switches* is known always to be a Boolean matrix indexed by an integer range, and so is not refined further.

Recall that, to represent the set correctly, the occurrence matrix must contain at most n (*capacity* in the *ring* example) 'true' entries, and that the explicit matrix must contain distinct entries. These constraints form part of a *local model*, associated with each element of the set produced by ρ. A local model is a list of declarations, constraints and *annotations* that hold only in the context of a particular refinement. Declarations and constraints are introduced into the local model with the keywords `letting` and `such that` respectively. Notice that, since these constraints are stated on the intermediate representation S', they must be refined before being added to the local model associated with S''.[2]

Annotations are used to add descriptive information to a refinement that will aid in the generation of the final model. Herein, we use the annotations `represent` and `symmetry`. `represent` records the relationship between an input expression and its refinement. It is essential for relating the objects in an

[2] The reader may wonder why in both clauses the constraint and S' are refined separately since, by compositionality, refining the constraint on S' must also refine S' itself. This is true, but then the refinements of S' are dictated by the representational needs of the constraint. By refining them separately, we obtain all refinements of S' and, where the refinement of S' individually does not match that obtained for the constraint, channelling constraints (see Section 7) are used to maintain consistency.

ESSENCE′ model to the objects in the corresponding ESSENCE specification, and for channelling between alternative representations of the same abstract object (see Section 7). In BOUNDEDSET1, the `represent` annotation describes the relationship between S and S', namely that S' is an occurrence representation (denoted by the keyword '*occset*') of S. Similarly, in BOUNDEDSET2, the `represent` annotation records that the pair $\langle S', Switches \rangle$ are a variable-sized explicit representation (denoted by the keyword '*varexpset*') of S.

Refinement introduces symmetry when it distinguishes between objects that were indistinguishable in the abstract specification. For example, when refining a set into a matrix, we distinguish the indices of the elements where the set did not. At present this is the only kind of symmetry that we consider, that is *index* symmetry in a matrix. This symmetry, which is very common [5], means that, say, the rows and/or columns of a 2-d matrix may be permuted in any (non-)solution to give a (non-)solution. It also commonly arises among the planes of matrices of higher dimensions. The `symmetry` *indexsym*(*matrix*, 1) annotation, as used in BOUNDEDSET2, records the fact that *matrix* has symmetry of its first index. Note that when the first argument is a tuple, as in the example, a permutation must be applied to all elements of the tuple simultaneously. Detecting symmetry as we introduce it is a considerably cheaper operation than symmetry detection on a finished model. In future, we intend to extend our system of annotations to include other types of symmetry, such as partial index symmetry and value symmetry.

As noted, a local model is associated with every element of the set produced by ρ. When ρ is applied recursively, the local model associated with the result is appended to the current local model. In BOUNDEDSET, for example, we would expect a `represent` annotation relating S' and S'' to be returned along with S''. For clarity, this operation is not shown explicitly — unless the returned local model is modified before being appended, as we will see.

We may now examine the result of applying BOUNDEDSET to the *ring* variable. Since *ring* is not nested, BOUNDEDSET produces two refinements:

BOUNDEDSET1	BOUNDEDSET2
ring′:`matrix (indexed by` *Nodes*`) of bool`	⟨*ring′*:`matrix (indexed by` 1..*capacity*`) of` *Nodes*,
	Switches:`matrix (indexed by` 1..*capacity*`) of bool`⟩
`such that` sum(*ring′*) ≤ *capacity*	`such that` AllDifferent(*ring′*)
`represent` *ring* `by` *occset*(*ring′*)	`represent` *ring* `by` *varexpset* (⟨*ring′*, *Switches* ⟩)
	`symmetry` *indexsym* (⟨*ring′*, *Switches*⟩, 1)

4.2 Refining an ESSENCE Specification

Given an ESSENCE specification, refinement proceeds constraint-wise. That is, each constraint and the objective function are refined in turn and the results composed to produce the final model. We illustrate this process by refining the MicroSonet1 specification given in Figure 4, which simply asks for a ring with a number of nodes installed not equal to 4.

Refinement begins with the expression $\rho(|ring| \neq 4)$, necessitating the use of the DISEQUATION rule. For space reasons, the details of this rule are omitted. Essentially, it refines both the left- and right-hand sides of the disequation and specifies that the refined expressions are not equal. $\rho(4)$ trivially produces 4,

```
given      nnodes:nat, capacity:nat
letting    Nodes be 1..nnodes
find       ring:set (maxsize capacity) of Nodes
such that  |ring| ≠ 4
```

Fig. 4. ESSENCE Specification of the MicroSonet1 problem.

hence we focus on the refinement of $\rho(|ring|)$ via the BOUNDEDCARDINALITY rule (Figure 5). This rule demonstrates the principle of selecting an appropriate refinement of an outer expression according to the refinement of an inner expression: knowledge of the representation of the set is a prerequisite for being able to count its elements. The selection is performed in both clauses by accessing, via the `with` operator, the local model associated with the refinement of S and predicating the success of the refinement on a particular `represent` annotation being present. Note that accessing the returned local model in this way does not affect its being appended to the current local model, as described above.

The two ESSENCE′ models for the MicroSonet1 problem are given in Figure 6. We now discuss how these models are constructed, given the refined constraints. The `given` and `find` statements are generated by starting from an object in the ESSENCE specification and tracing the `represent` statements associated with the refinement of each constraint to find the corresponding ESSENCE′ object. The `given` statements in MicroSonet1 are already in ESSENCE′, so no change is necessary. In the first model in Figure 6, the refinement of the single constraint produced the annotation `represent` *ring* `by` *occset*(*ring*′). Since *ring* appears in a `find` statement of the specification, *ring*′ is added to the `find` statement of the model. Similarly, the annotation `represent` *ring* `by` *varexpset*($\langle ring', Switches\rangle$) is used to generate the `find` statements in the second model. Note that this same mechanism can be used to express a solution to a model in the terms of the original specification.

The `letting` statements in a model contain the union of the `letting` statements in the refinement of each constraint in the specification. `letting` statements in the specification and already in ESSENCE′ are also included in the model, if used (e.g. *Nodes* in the example). The constraints in a model consist primarily of the refinements of each constraint in the specification. Constraints

BOUNDEDCARDINALITY1 $\rho(|S$:`set (maxsize` n`) of` $\tau|) \overset{\text{ref}}{\rightarrow}$

```
{ sum(S′)                                          | S′:matrix (indexed by _) of bool
  |
  (S′ with represent S by occset(S′))∈ ρ(S)        |
}
```

BOUNDEDCARDINALITY2 $\rho(|S$:`set (maxsize` n`) of` $\tau|) \overset{\text{ref}}{\rightarrow}$

```
{ sum(Switches)                                    | S′:matrix (indexed by 1..n) of _
  |                                                | Switches:matrix (indexed by 1..n) of bool
  (⟨S′, Switches⟩ with
   represent S by varexpset(⟨S′, Switches⟩)) ∈ ρ(S) |
}
```

Fig. 5. The BOUNDEDCARDINALITY refinement rule.

```
given      nnodes:nat, capacity:nat
letting    Nodes be 1..nnodes
find       ring′:matrix (indexed by Nodes) of bool
such that sum(ring′) ≤ capacity
          sum(ring′) ≠ 4
```

```
given      nnodes:nat, capacity:nat
letting    Nodes be 1..nnodes
find       ring′:matrix (indexed by 1..capacity) of Nodes
           Switches:matrix (indexed by 1..capacity) of bool
such that AllDifferent(ring′)
          sum(Switches′) ≠ 4
symmetry   indexsym (⟨ring′, Switches⟩, 1)
```

Fig. 6. Alternative Essence′ models of the MicroSonet1 problem.

```
given      nrings:nat, nnodes:nat, capacity:nat
letting    Nodes be 1..nnodes
find       rings: mset (size nrings) of set (maxsize capacity) of Nodes
minimising ∑r∈rings |r|
```

Fig. 7. Essence Specification of the MicroSonet2 problem.

introduced in the local model of the refinement of a constraint are also included, such as $\text{sum}(ring') \leq capacity$ and $\text{AllDifferent}(ring')$ in the two models of Figure 6.

The final step is to make use of the `symmetry` annotations on the model to add appropriate symmetry-breaking constraints, or employ a dynamic method such as SBDS [9] or SBDD [4]. Then, as discussed in Section 3.1, a straightforward translation is performed from the generic Essence′ language to the input language of an established constraint toolkit.

5 Refining Nested Types

Thus far, our examples have not covered the refinement of nested type expressions. This is one of the most important features of Conjure and is discussed in this section. Consider the specification of MicroSonet2, given in Figure 7, which is the Sonet problem with the constraint removed. Refinement begins with the expression $\rho(\sum_{r \in rings} |r|)$ via the MultisetSum rule (Figure 8). The first step in refining this expression, which quantifies over the set *rings*, is to refine the binding expression, $r \in rings$, which is, in turn, dependent on the refinement of the multiset variable *rings*. The clause of MultisetSum given in the figure chooses an explicit matrix representation for the multiset. The returned expression therefore quantifies over the indices of this matrix (Essence′ allows quantification over integer ranges only).

Given the explicit representation of the multiset variable over which quantification takes place, say *rings*′ in the example, the arithmetic expression, $|r|$ here, is refined. In doing so, *rings*′, indexed by the Essence′ quantified variable j, is subsituted for the abstract quantified variable r (using $\alpha[i \mapsto S'[j]]$ in the rule).

The statements of the local model associated with this refinement must also be quantified to hold for each possible value of j. This operation is performed by matching the whole of the local model returned, adding the quantification, and explicitly `append`ing it to the current local model. This is the exception to the default of appending a returned local model to the current local model intact.

In the example MULTISETSUM produces $\rho(|rings'[j]|)$. It is at this point that the power of compositional refinement of nested types becomes clear. Recall the refinement of the MicroSonet specification in Section 4 where $|ring|$ was refined. Having decomposed the original summation above, the remaining refinement proceeds in much the same fashion as the corresponding part of the MicroSonet1 refinement. Both representations of *ring* are of course generated to give alternative ESSENCE′ models. For brevity, however, Figure 9 presents only the one based on the occurrence representation.

Note that $rings'$ is not present in the model. This is an intermediate representation since, although more concrete than *rings* itself, it is a matrix of sets, and therefore not an ESSENCE′ object. During refinement of MicroSonet2, the following pair of annotations are added: `represent` *rings* `by` *expmset*($rings'$) and $\forall j$:1..*nrings* `represent` $rings'[j]$ `by` *occset*($rings''[j]$).[3] Generalising the process described in Section 4.2 to following a *chain* of annotations, `find` statements are generated for the ESSENCE′ variables only. Similarly, the `symmetry` annotation produced during refinement is `symmetry` *indexsym*($rings'$, 1). We use the annotation relating $rings'$ and $rings''$ given above to produce the final annotation, `symmetry` *indexsym*($rings''$, 1).

6 Refining the Sonet Problem

We now perform a full refinement of the Sonet specification in Figure 2. In the previous section, we saw the refinement of the objective function, which leaves only $\forall pair \in demand.\exists r \in rings.pair \subseteq r$. Refinement begins with $\rho(\forall pair \in demand.\exists r \in rings.pair \subseteq r)$ using the FORALLSET rule (Figure 10). This rule works in much the same way as MULTISETSUM, described above.

[3] `genSymbol`() respects the index structure of its first argument. Hence, given the indexed one-dimensional array $rings'[j]$:`set` (`maxsize` *capacity*) `of` *Nodes*, `genSymbol`($rings'[j]$, `matrix` (`indexed by` *Nodes*) `of bool`) produces a partially indexed two-dimensional array, $rings''[j]$:`matrix`(`indexed by` *Nodes*) `of bool`.

MULTISETSUM1 $\rho(\sum_{i:\tau \in S:\texttt{mset(size } n\texttt{) of } \tau} \alpha{:}lb..ub) \overset{\text{ref}}{\rightarrow}$

{ $\text{sum}_j.\alpha'$
`represent` S `by` *expmset*(S')
`symmetry` *indexsym*(S', 1)
`append` $\forall j{:}1..n.\Gamma$
|
$(\alpha'$ `with local model` $\Gamma) \in \rho(\alpha[i \mapsto S'[j]])$
}

j = `genSymbol`$(i, 1..n)$
S' = `genSymbol`$(S,$ `matrix (indexed by` $1..n$`) of` $\tau)$
$\alpha'{:}lb..ub$

Fig. 8. A clause of the MULTISETSUM rule.

```
given       nrings:nat, nnodes:nat, capacity:nat
letting     Nodes be 1..nnodes
find        rings''matrix (indexed by 1..nrings, Nodes) of bool
minimising  sum_{j:1..nrings}(rings''[j])
such that   ∀j:1..nrings.(sum(rings''[j]) ≤ capacity)
symmetry    indexsym (rings'', 1)
```

Fig. 9. ESSENCE′ model of the MicroSonet2 Specification.

In this case, the rule produces $\rho(\exists r \in rings.demand'[j] \subseteq r)$, necessitating the use of the MULTISETEXISTS rule (Figure 11), whose operation also follows the same pattern. Here, MULTISETEXISTS produces the partially-refined expression $\rho(demand'[j] \subseteq rings'[k])$, which requires the SUBSETBOUNDEDSET rule (Figure 12). In the example, the expressions produced by either clause of the rule are simple to refine and so not described in detail. If we follow the first clause of this rule and refine the objective function as in the previous section, the ESSENCE′ model given in Figure 13 is produced.

Despite the fact that refining both the objective function and the constraint in the way described produces a two-dimensional Boolean matrix, notice that the model contains only a single matrix of decision variables, *rings*″. From the annotations, it is clear that the matrices produced by the two refinements are identical representations of *rings*, and so one of the pair may be substituted for all occurrences of the other.

The refinement of the *demand* relation to a two-dimensional matrix indicates how to translate the input data as originally specified into a form easily processed by a constraint toolkit. Note that, when refining a decision variable in this way, an annotation indicating symmetry of the first index is added. However, since *demand* is constant at solution time, this symmetry is ignored at present.

The translation of $\forall j : 1..m.\ \exists k : 1..nrings.\ \forall a : 1..2.\ rings''[k][demand''[j][a]]$ from ESSENCE′ to an existing constraint language, although at first sight complex, is relatively straightforward. The outer universal quantifier can be translated to a 'for' loop, or equivalent iterator, ranging over the integers $1..m$. The existential quantifier can then be represented as iterated disjunction, each disjunct consisting of a conjunction of two cases representing the inner universal quantifier. This, however, is not a very efficient representation of the demand constraint. It is essentially a long-winded version of a scalar product constraint on each pair of rows denoting nodes related by *demand*, which is how this con-

```
ForallSet ρ(∀i:τ ∈ S:set (size n) of τ.φ:bool) ~ref~>
{ ∀j.φ'                                                  | j = genSymbol(i, 1..n)
  represent S by expset(S')                              | S' = genSymbol(S, matrix (indexed by 1..n) of τ)
  symmetry  indexsym(S', 1)
  append    ∀j:1..n.Γ
  |
  (φ' with local model Γ) ∈ ρ(φ[i ↦ S'[j]])
}
```

Fig. 10. A clause of the FORALLSET rule.

MultisetExists $\rho(\exists i{:}\tau \in S{:}\texttt{set (size } n\texttt{) of } \tau.\phi{:}\texttt{bool}) \overset{\text{ref}}{\rightarrow}$

{ $\exists j.\phi'$	$j = \texttt{genSymbol}(i, 1..n)$
represent S by $expset(S')$	$S' = \texttt{genSymbol}(S, \texttt{matrix (indexed by } 1..n\texttt{) of } \tau)$
symmetry $indexsym(S', 1)$	$\phi'{:}\texttt{bool}$
append $\forall j.\Gamma$	
|	
ϕ' with local model $\Gamma \in \rho(\phi[i \mapsto S'[j]])$	
}	

Fig. 11. A clause of MultisetExists rule.

SubsetBoundedSet1 $\rho(S_1{:}\texttt{set (size } n\texttt{) of } \tau \subseteq S_2{:}\texttt{set (maxsize } p\texttt{) of } \tau) \overset{\text{ref}}{\rightarrow}$

{ ϕ	$a = \texttt{genSymbol}(\text{`quantified variable'}, 1..n)$
such that $\chi \wedge \psi$	$S'_1 = \texttt{genSymbol}(S_1, \texttt{matrix (indexed by } 1..n\texttt{) of } \tau)$
represent S_1 by $expset(S'_1)$	$S'_2 = \texttt{genSymbol}(S_2, \texttt{matrix (indexed by } \tau\texttt{) of bool})$
represent S_2 by $occset(S'_2)$	$\phi{:}\texttt{bool}$
symmetry $indexsym(S'_1, 1)$	$\chi{:}\texttt{bool}$
|	$\psi{:}\texttt{bool}$
$\phi \in \rho(\forall a.S'_2[S'_1[a]])$	
$\chi \in \rho(\text{sum}(S'_2) \leq p)$	
$\psi \in \rho(\text{AllDifferent}(S'_1))$	
}	

SubsetBoundedSet2 $\rho(S_1{:}\texttt{set (size } n\texttt{) of } \tau \subseteq S_2{:}\texttt{set (maxsize } p\texttt{) of } \tau) \overset{\text{ref}}{\rightarrow}$

{ ϕ	$a = \texttt{genSymbol}(\text{`quantified variable'}, 1..n)$
such that $\chi \wedge \psi$	$b = \texttt{genSymbol}(\text{`quantified variable'}, 1..p)$
represent S_1 by $expset(S'_1)$	$S'_1 = \texttt{genSymbol}(S_1,$
represent S_2 by $varexpset(\langle S'_2, Switches\rangle)$	$\quad \texttt{matrix (indexed by } 1..n\texttt{) of } \tau)$
symmetry $indexsym(S'_1, 1)$	$S'_2 = \texttt{genSymbol}(S_2,$
$\quad indexsym(\langle S'_2, Switches\rangle, 1)$	$\quad \texttt{matrix (indexed by } 1..p\texttt{) of } \tau)$
|	$Switches = \texttt{genSymbol}(S_2,$
$\phi \in \rho(\forall a \exists b.((S'_1[a] = S'_2[b]) \wedge Switches[b]))$	$\quad \texttt{matrix (indexed by } 1..p\texttt{) of bool})$
$\chi \in \rho(\text{AllDifferent}(S'_1))$	$\phi{:}\texttt{bool}$
$\psi \in \rho(\text{AllDifferent}(S'_2))$	$\chi{:}\texttt{bool}$
}	$\psi{:}\texttt{bool}$

Fig. 12. Two clauses of the SubsetBoundedSet rule.

straint has been modelled by hand previously [7]. Recognising this fact and performing the appropriate transformation to an efficient representation is the domain of *transformation* rules, as discussed in [8].

7 Generating Channelling Constraints Automatically

Good CSP models frequently contain two or more representations of the same abstract variable. The ability to support multiple representations is a powerful modelling tool, since different constraints can typically be stated most straightforwardly on different representations. This is one of the main reasons that we generate multiple concrete representations for each abstract variable. To be able to use different representations within the same constraint program it is necessary to introduce *channelling* constraints [3] between the alternatives to ensure that they remain consistent.

Reconsider the Sonet problem. For simplicity, the example model discussed in the previous section was deliberately chosen so that both the objective and the constraint are stated on the same *rings*$''$ matrix. Hence, channelling was not necessary. If, however, we maintain the refinement of the objective and consider

```
given       nrings:nat, nnodes:nat, capacity:nat
letting     Nodes be 1..nnodes
given       demand'':matrix (indexed by 1..m, 1..2) of Nodes
find        rings'':matrix (indexed by 1..nrings, Nodes) of bool
minimising  sum_{j:1..nrings}(rings''[j])
such that   ∀j:1..nrings.(sum(rings''[j]) ≤ capacity)
            ∀j:1..m. ∃k:1..nrings. ∀a:1..2. rings''[k][demand''[j][a]]
symmetry    indexsym (rings'', 1)
```

Fig. 13. ESSENCE′ model of the Sonet specification.

```
given       nrings:nat, nnodes:nat, capacity:nat
letting     Nodes be 1..nnodes
given       demand'':matrix (indexed by 1..m, 1..2) of Nodes
find        rings''_1:matrix (indexed by 1..nrings, Nodes) of bool
            rings''_2:matrix (indexed by 1..nrings, 1..capacity) of Nodes
            Switches_2:matrix (indexed by 1..nrings, 1..capacity) of bool
minimising  sum_{j:1..nrings}(rings''_1[j])
such that   ∀j:1..nrings.(sum(rings''_1[j]) ≤ capacity)
            ∀j:1..m ∃k:1..nrings ∀a:1..2 ∃b:1..capacity
              (demand''[j][a] = rings''_2[k][b]) ∧ Switches_2[k][b]
            ∀i ∈ 1..nrings ∀j ∈ Nodes
              rings''_1[i][j] ↔ (∃k ∈ 1..capacity rings''_2[i][k] = j ∧ Switches[i][k])
symmetry    indexsym (rings''_1, 1)
            indexsym (⟨rings''_2, Switches_2⟩, 1)
```

Fig. 14. Channelling ESSENCE′ model of the Sonet specification.

an alternative refinement of $\forall pair \in demand.\exists r \in rings.pair \subseteq r$, where each ring is represented explicitly, the resulting model is presented in Figure 14. Since this model contains two representations of *rings*, $rings''_1$ (occurrence) and $rings''_2$ (variable-sized explicit generated via SUBSETBOUNDEDSET2), channelling constraints are necessary to keep the two consistent.

The mechanism for generating channelling constraints again relies upon the annotations. The three `represent` annotations used herein, *varexpset*,*occset* and *expset*, have corresponding axioms as follows (similarly for multisets):

1. `represent` X:`set (maxsize` n`) of` τ `by` $varexpset(\langle Y$:`matrix (indexed by` $1..n$`) of` τ, $Switches$:`matrix (indexed by` $1..n$`) of bool`$\rangle)$
   corresponds to $\forall i \in \tau(i \in X \leftrightarrow \exists j \in 1..n(Y[j] = i \wedge Switches[j]))$
2. `represent` X:`set (size` n`) of` τ `by` $expset(Y$:`matrix (indexed by` $1..n$`) of` $\tau)$
   corresponds to $\forall i \in \tau(i \in X \leftrightarrow \exists j \in 1..n(Y[j] = i))$

3a. `represent` X:`set (maxsize` n`) of` τ `by` $occset(Y$:`matrix (indexed by` τ`) of bool)`
   corresponds to $\forall i \in \tau(i \in X \leftrightarrow Y[i])$

3b. `represent` X:`set (size` n`) of` τ `by` $occset(Y$:`matrix (indexed by` τ`) of bool)`
   corresponds to $\forall i \in \tau(i \in X \leftrightarrow Y[i])$

Both the refinement of the objective and the constraint in the example share the intermediate representation *rings*′. This is related to *rings* by the annotation `represent` *rings* `by` $expmset(rings')$, therefore the following axiom (2) applies: $\forall i \in$ `set (maxsize` *capacity*`) of` $Nodes(i \in rings \leftrightarrow \exists j \in 1..nrings(rings'[j] = i))$. The annotation relating *rings*′ and $rings''_1$, from the refinement of the objective, is $\forall j{:}1..nrings$ `represent` $rings'[j]$ `by` $occset(rings''_1[j])$. Hence, axiom (3a) applies: $\forall j{:}1..nrings\ \forall i \in Nodes(i \in rings'[j] \leftrightarrow rings''[j][i])$. Now we can link *rings* and $rings''_1$ directly:

$\forall i \in$ `set (maxsize` *capacity*`) of` *Nodes*
$i \in rings \leftrightarrow (\exists j \in 1..nrings\ \forall k \in Nodes\ k \in i \leftrightarrow rings''_1[j][k])$

The annotation relating $rings'$ and $rings''_2$, from the refinement of the constraint, is $\forall j$:1..*nrings* `represent` $rings'[j]$ `by` $varexpset(\langle rings''_2[j],\ Switches_2\rangle)$. Hence, axiom (1) applies:

$\forall j$:1..*nrings* $\forall i \in Nodes$
$i \in rings'[j] \leftrightarrow \exists k \in 1..capacity(rings''_2[j][k] = i \wedge Switches_2[j][k])$

Similarly, we can link *rings* and $rings''_2$ directly:

$\forall i \in$ `set (maxsize` *capacity*`) of` *Nodes*
$i \in rings \leftrightarrow (\exists j \in 1..nrings\ \forall k \in Nodes\ k \in i \leftrightarrow$
$(\exists l \in 1..capacity\ rings''_2[j][l] = k \wedge Switches_2[j][l]))$

Finally, having channelled between *rings* and $rings''_1$ and between *rings* and $rings''_2$, we can substitute to get the required channelling constraint between $rings''_1$ and $rings''_2$, as presented in Figure 14:

$\forall i \in 1..nrings\ \forall j \in Nodes$
$rings''_1[i][j] \leftrightarrow (\exists k \in 1..capacity\ rings''_2[i][k] = j \wedge Switches[i][k])$

8 Issues in Language Design

Having seen how CONJURE operates, we can consider how the demands of effective refinement have driven the design of ESSENCE. In general these demands have led to a language that is more expressive and contains constructs that are more abstract than previous specification languages such as OPL [16] and ESRA [6], each of which was designed to be more abstract than its predecessors.

If the language is insufficiently abstract, specification will involve making modelling decisions. Hence, alternatives to these pre-made decisions cannot be considered during refinement. Many languages, such as OPL, do not support sets or relations, so the problem specifications must employ matrices. Though ESRA supports sets and relations, it does not support sets of sets, so a specification cannot contain a set of sets as an alternative to certain types of relation. In these languages symmetry is introduced by, and must be identified by, the user.

A similar principle is that the language should not force irrelevant information to be provided in a specification. For example, ESSENCE enables an enumerated type to be specified without naming the individuals in the type. This feature is frequently useful. For example, in the Golfers problem there is no need to distinguish the golfers. In constraint satisfaction problems the variables and values are always named, so the refinement system must introduce names for the golfers. In doing so, the refinement system recognises that it is introducing symmetry into the problem and records this in an annotation.

This discussion shows that the development of a compositional refinement system places major demands on the specification language. Hence, we have developed the language and refinement system in tandem. This has resulted in a radically different and significantly more abstract language than its predecessors.

9 Related Work in Modelling Support

As well as OPL, ESRA and Fiona, there are several other methods to aid in constraint modeling. We discuss some of them briefly.

NP-SPEC allows the specification of any NP-complete problem. However, specifications are given in second order logic. Refinement from an abstract specification into NP-SPEC is performed by hand. Renker et al.[12] discuss representing abstract specifications in Z [15], an expressive abstract language. Tools for Z language can check if two representations of a specification are equivalent. However, refinement and transformation of the Z specification is also manual.

KIDS [13] is a domain-independent system that supports the interactive refinement of formal specifications into executable programs via a series of rewrite rules. Producing programs instead of models introduces complexity, since each statement may have side-effects. This renders large-scale transformations of the specification, as discussed in this paper, very difficult to prove correct.

Other work has extended constraint languages to support sets and multisets directly [10, 17]. However, these extensions do not support nested types and so cannot model the abstract specifications we have considered herein directly.

10 Conclusion

Essence and Conjure are a major step towards our ultimate goal, the automation of modelling. Achieving this goal requires identifying and automating heuristics for choosing the most effective models from among the alternatives generated.[4] Few such heuristics are known to constraint modelling. The work presented here should be valuable in the search for the heuristics as it lays out precisely what it is that a heuristic must choose among. It also raises new important questions. For example, is the best way of representing a decision variable whose type is "set of X" (where X is an arbitrary type) constructed from the best way of representing X? Does the best way of breaking symmetry in "set of X" involve the best way of breaking symmetry in X?

Acknowledgements Ian Miguel is supported by a UK-Royal Academy of Engineering/EPSRC Post-doctoral Research Fellowship. We thank Adam Bakewell for contributing to the previous version of this work, Brahim Hnich and Toby Walsh for useful discussions, and our anonymous reviewers for their insightful comments.

References

1. A. Bakewell, A.M. Frisch and I. Miguel. Towards Automatic Modelling of Constraint Satisfaction Problems: A System Based on Compositional Refinement *Proc. 2nd International Workshop on Modelling and Reformulating CSPs*, 2–17, 2003

[4] Corresponding to the distinction linguists make between theories of competence and performance.

2. M. Cadoli, L. Palopoli, A. Schaerf and D. Vasile. NP–SPEC:An Executable Specification Language for Solving All Problems in NP. *Proc. 1st International Workshop on Practical Aspects of Declarative Languages*, LNCS 1551, 16-30, 1999.
3. B.M.W. Cheng, K.M.F. Choi, J.H. Lee, J.C.K. Wu. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints* 4(2), 167-192, 1999.
4. T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *Proc. 7th International Conference on Principles and Practice of Constraint Programming*, 93-107, 2001.
5. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh. Breaking row and column symmetries in matrix models. *Proc. 8th International Conference on Principles and Practice of Constraint Programming*, 462-476, 2002.
6. P. Flener, J. Pearson and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems, *Proc. LOPSTR'03: Revised Selected Papers*, LNCS 3018, 2004.
7. A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, T. Walsh. Transforming and Refining Abstract Constraint Specifications. *Proc. CSCLP'04: Joint Annual Workshop of ERCIM/Colognet on Constraint Solving and Constraint Logic Programming*, 2004.
8. A.M. Frisch, I. Miguel and T. Walsh. CGRASS: A System for Transforming Constraint Satisfaction Problems. *proc. ERCIM/CologNet workshop on Constraint Logic Programming.* (LNAI 2627), 15-30, 2002.
9. I.P. Gent and B.M. Smith. Symmetry breaking during search in constraint programming. *Proc. European Conference on Artificial Intelligence*, 599-603, 2000.
10. C. Gervet. Conjunto: constraint logic programming with finite set domains. *Proc. International Symposium on Logic Programming*, 339-358, 1994.
11. B. Hnich. *Function Variables for Constraint Programming.* PhD thesis, Uppsala University, 2004.
12. G. Renker and H. Ahriz. Building Models Through Formal Specification. *Proc. International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, LNCS 3011, 395-401, 2004.
13. D.R. Smith. KIDS:A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024-1043,1990.
14. B.M. Smith, K. Stergiou, T. Walsh. Modelling the golomb ruler problem. *Proc. IJCAI-99 Wshop on Non-binary Constraints*, 1999.
15. J.M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., 1989
16. P. Van Hentenryck. *The OPL optimization programming language.* MIT Press,1999.
17. T. Walsh. Consistency and propagation with multiset constraints: a formal viewpoint. *Proc. 9th Int. Conf. on Principles and Practice of Constraint Programming*, 724-738, 2003.