The Rules of Constraint Modelling

Alan M. Frisch and **Chris Jefferson** and **Bernadette Martínez Hernández**
Artificial Intelligence Group, Dept. of Computer Science, Univ. of York, UK
{frisch,caj,berna}@cs.york.ac.uk

Ian Miguel
School of Computer Science, Univ. of St Andrews, UK
ianm@dcs.st-and.ac.uk

Abstract

Many and diverse combinatorial problems have been solved successfully using finite-domain constraint programming. However, to apply constraint programming to a particular domain, the problem must first 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* a specification of a problem in the abstract constraint specification language ESSENCE into a set of alternative constraint models. Refinement is *compositional*: alternative constraint models are generated by composing refinements of the components of the specification. Experimental results demonstrate that CONJURE is able to generate a variety of models for practical problems from their ESSENCE specifications.

1 Introduction

To employ finite-domain constraint programming technology to solve a problem, the problem first must be characterised, or *modelled*, by a set of constraints on decision variables that solutions must satisfy. Modelling can be difficult and requires expertise, thus limiting widespread use of constraint technology. The vast majority of research on constraint modelling 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. Doing this requires 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.

The rules are embedded in the CONJURE system, which, given an ESSENCE specification, generates models of the type supported by existing constraint solvers. Hence, we refer to our rules as *refinement rules*. Our current focus is on generating a set of correct models that includes those that a human expert would generate. Future work will focus on generating only *good* models.

2 Challenges and Contributions

A central task, arguably *the* central task, of modelling most combinatorial problems is choosing a representation of complex decision variables. Current finite-domain constraint solvers provide decision variables whose domains contain atomic elements, which we call atomic variables.[1] Yet, combinatorial problems often require finding a more complex combinatorial structure. For example, the Social Golfers Problem (SGP, see [9] problem 10) requires partitioning a set G of golfers in each week of play. Thus, the goal is to find a multiset of partitions of G, i.e. a multiset of sets of sets of G. Modelling the SGP requires deciding how to represent this complex decision variable as a constrained collection of atomic variables.

In concert with choosing a representation of complex decision variables is the task of representing the constraints of the problem. In its natural form, a combinatorial problem imposes constraints on the combinatorial structure that is sought. These constraints must be "translated" so that they are imposed on the representation of the decision variables.

This paper shows how these two central tasks of modelling can be formalised and automated, and in doing so reports two principal contributions. First, we have designed a language, called ESSENCE, that enables combinatorial problems to be stated at a high level of abstraction. This level of abstraction is a consequence of three features: (1) The language supports a wide range of types (including sets, multisets, relations, functions, partitions) and decision variables can be of these types. (2) All types can be nested to arbitrary depth; for example, a decision variable can be of type set, set of sets, set of set of sets, and so forth. (3) Constraints can contain quantifiers that range over decision variables. For example, if a decision variable X is of type set of sets, a constraint can

[1] Some finite-domain constraint solvers support variables whose domain elements are finite sets of atomic elements. Though all aspects of our work have considered this, for the sake of simplicity this paper pretends such variables do not exist.

be of the form $\forall x \in X.\phi$. The ESSENCE language is outlined in Sec. 3, though only in enough detail to enable the presentation of our second principal contribution.

Our second major contribution is the formulation and automation of a set of rules that can refine constraints on complex variables in an ESSENCE specification into constraints on atomic variables, which is the level of abstraction provided by existing constraint languages and toolkits. Our formal rules are presented in Sec. 4.

An attempt to formulate such a set of refinement rules confronts two primary difficulties and many secondary ones. Before proceeding it is worth considering the primary ones.

The first difficulty arises because expressions, particularly decision variables of non-atomic type, can usually, if not always, be refined in multiple ways. Furthermore, the refinement of an operator depends on how its operands are refined. For example, in refining the constraint $S_1 = S_2$, where S_1 and S_2 are atomic set variables, the treatment of equality is different for every combination of ways that S_1 and S_2 can be refined. Indeed, it is possible that an operator is inapplicable to certain refinements of its operands.

The second major difficulty arises from arbitrary nesting of types. Refining an operator that is applied to expressions whose types are nested provides the biggest challenge. Consider refining the constraint $A = B$, where A and B are decision variables of some type τ that is nested arbitrarily deep. The generated constraint must involve all components of both A and B. Since there is no bound on the nesting of τ, this complex constraint would have to be generated through recursive rule applications, none of which can look arbitrarily deep into the nesting of τ. Furthermore, we wish to produce refinements in which A and B do not have the same kind of representation. An especially tricky case is when a quantifier ranges over a decision variable of a nested type, such as $\forall x \subseteq A\ \phi$, where A is as above.

Modelling in constraints involves more than just representing decision variables and problem constraints. Constraint models often contain many symmetries, often enormous numbers of them, which result in redundancies in the search space. Expert modellers are able to identify such symmetries 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. An automated modelling system ought to identify the symmetry that it introduces into a model. Sec. 5 explains how this can be integrated into our architecture for model generation.

Another technique used by expert modellers is to represent a complex decision variable with multiple representations simultaneously and impose channelling constraints to keep the representations consistent with each other [4; 8; 11]. This sometimes yields more propagation, and therefore reduced search, than a single representation. As will be seen, the rules of CONJURE generate models with multiple representations. Sec. 6 explains how our refinement rules can generate the information that is needed to automatically generate the appropriate channelling constraints.

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

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

Figure 1: ESSENCE specification of the Golomb Ruler problem.

We conjecture that the good models of a problem can be generated automatically by formulating the problem in ESSENCE and then refining the specification using rules of the kind presented in this paper. We have implemented a program, called CONJURE, that currently refines a subset of ESSENCE, called mini-ESSENCE. This gives us a platform with which we have experimentally tested our conjecture, as reported in in Sec. 7.

3 An Introduction to ESSENCE

This section briefly introduces the abstract specification language ESSENCE. It is an evolving and growing language; version 1.0 is used within this paper. A full specification of the language can be found at `www.cs.york.ac.uk/aig/constraints/AutoModel/`.

Let us begin by considering the specification of the Golomb Ruler Problem (GRP, problem 6 at www.csplib.org), shown in Fig. 1). A specification is a list of statements, of which there are seven kinds, signalled by the keywords `given`, `where`, `letting`, `find`, `maximising`, `minimising` and `such that`. Statements are composed into specifications according to the regular expression

$$(\texttt{given} \mid \texttt{letting} \mid \texttt{where})^* \ \texttt{find}^+ \ [\texttt{minimising} \mid \texttt{maximising}] \ (\texttt{such that})^*$$

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 other modelling languages, they are provided elsewhere. "Where" statements impose restrictions on the parameter values; only parameter values meeting the restrictions specify a problem instance. "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, restricting it to be non-negative and declaring $bound$ 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 and subexpression has a type. The type of the decision variable $Ticks$ 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 atomic types `int` (integer), `bool` (Boolean) and $l..u$ (an integer range type), where l and u 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, set of integer, set of set of integer, and so forth. If τ is a type, α is an integer expression, and $I_1, \ldots, I_n$ are expressions of type range then the following are types that appear in this paper:

`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

Also among the types of ESSENCE are relations between any two types, partial and total functions from any type to any type, partitions of a set of any type and permutations of any type.

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 (e.g., integer range but not integer). The constraint in the GRP can be paraphrased 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 $\{i, j\} \subseteq Ticks$ means that two distinct elements are drawn from $Ticks$ and, without loss of generality, one is called i and the other is called j.

Now consider the specification of the Sonet problem shown in Fig. 2. Notice that *Nodes* is declared to be a range. 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 objective is to minimise the sum of the number of nodes installed on each ring. The constraint ensures that any pair of nodes that must communicate are installed on a common ring.

The Target Language: ESSENCE′

ESSENCE specifications are refined into a target language called ESSENCE′, which is a subset of ESSENCE with a level of abstraction similar to that of existing constraint languages, OPL [19] being the closest. The only types ESSENCE′ has are integers, integer ranges, Booleans and matrices; it does not have enumerated types or types of unnamed elements. Binders, such as quantifiers and summations, range only over integer ranges. From this generic constraint language, it is

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 [8])

`given` *nrings*:`int`, *nnodes*:`int`, *capacity*:`int`
`where` *nrings* ≥ 1, *nnodes* ≥ 1, *capacity* ≥ 1
`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` $\sum_{r \in rings} |r|$
`such that` $\forall pair \in demand . \exists r \in rings . pair \subseteq r$

Figure 2: ESSENCE specification of the Sonet problem.

a short step to an established constraint language, such as OPL, Solver, or Eclipse. To perform this step, we are developing a suite of back-end translators. In future, we also intend to translate to more restricted languages, such as SAT and Pseudo-Boolean formulations.

4 The Architecture of CONJURE

This section discusses the refinement rules CONJURE uses to refine an ESSENCE expression into a set of ESSENCE′ expressions. For concision we give only a small subset of all the refinement rules[2]; the remainder follow a similar pattern.

Refining a Simple ESSENCE Specification

Refinement of an ESSENCE specification begins by refining each constraint and the objective function in turn. Let us begin by considering the single-constraint MicroSonet$_1$ specification (Fig. 3), where the goal is to fill two rings with the same nodes.

A key consideration in refining MicroSonet$_1$ is the representation of each *ring*, each of which is a fixed-size set. This paper uses two representations of fixed-size sets. The *explicit representation* is a one-dimensional matrix of *Nodes* indexed by 1..*capacity*. Each element of the matrix corresponds to an element of the set. To represent the set properly, the elements of the matrix are constrained to take distinct values. The *occurrence representation* is a one-dimensional Boolean matrix indexed by *Nodes*, where a *true* entry indicates that the corresponding node is in the set. To represent the set properly, the number of *true* entries must be equal to *capacity* (for counting we treat *true/false* as 1/0). Fig. 3 gives ESSENCE′ models using both representations. Note that some symbols of ESSENCE, such as "$\forall$" and "Σ", can also be written textually, such as "forall" and "sum". In writing ESSENCE′ we use the textual version.

The refinement operator, ρ, is a function that maps every ESSENCE expression to a *set* of ESSENCE′ expressions. As explained later, each of these ESSENCE′ expressions is tagged with further information necessary to construct the

[2]See http://www.cs.york.ac.uk/aig/constraints/AutoModel/ for the complete set.

```
given       nnodes : int, capacity : int
where       nnodes ≥ 1, capacity ≥ 1
letting     Nodes be 1..nnodes
find        ring1 : set (size capacity) of Nodes
            ring2 : set (size capacity) of Nodes
such that   ring1 = ring2
```

```
given       nnodes : int, capacity : int
where       nnodes ≥ 1, capacity ≥ 1
letting     Nodes be 1..nnodes
find        ring'1 : matrix [indexed by 1.. capacity] of Nodes
            ring'2 : matrix [indexed by 1.. capacity] of Nodes
such that   forall (i : 1..capacity) exists (j : 1..capacity) ring'1[i] = ring'2[j]
            AllDifferent(ring'1)
            AllDifferent(ring'2)
```

```
given       nnodes : int, capacity : int
where       nnodes ≥ 1, capacity ≥ 1
letting     Nodes be 1..nnodes
find        ring'1 : matrix [indexed by Nodes] of bool
            ring'2 : matrix [indexed by Nodes] of bool
such that   forall (i : Nodes) ring'1[i]= ring'2[i]
            sum (i : Nodes) ring'1[i]= capacity
            sum (i : Nodes) ring'2[i]= capacity
```

Figure 3: ESSENCE specification and two ESSENCE′ models of the MicroSonet$_1$ problem.

final model. ρ is defined inductively by a set of uniquely-named equations 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 set comprehension. For perspicuity, each equation is split into *rules*, written: $R_1\ \rho(e) \overset{\text{ref}}{\to} e'_1$, $R_2\ \rho(e) \overset{\text{ref}}{\to} e'_2, \ldots,$ $R_n\ \rho(e) \overset{\text{ref}}{\to} e'_n$.

The SIZEDSETEQUALITY equation is responsible for refining expressions of the form $S_1 = S_2$, where both S_1 and S_2 are expressions of type fixed-size set. Fig. 4 gives two rules of this equation. The complete version of SIZEDSETEQUALITY gives one rule for for each combination of a representation of S_1 and a representation of S_2 and a constraint for imposing equality between the them.

To illustrate the operation of a rule, we now discuss how the SIZEDSETEQUALITY1 rule refines *ring*$_1$ = *ring*$_2$. SIZEDSETEQUALITY1 refines S_1 and S_2 into explicit one-dimensional matrices denoted by S'_1 and S'_2. The SIZEDSETEQUALITY2 rule proceeds similarly. The reader may wonder at the strategy of refining the arguments of the equality constraint inside the equality rule itself. This is explained in the next sub-section.

When a rule needs to introduce a new identifier, such as that denoted by S'_1, it does so by making use of the `genSymbol` function. This function takes two arguments. The first is either an explicit category (see Sec. 3), or an identifier from which the category information is copied. The second is the type of the new identifier, such as a one-dimensional matrix used for the explicit representation. The `genSymbol` function creates a new identifier of the required category and type that appears nowhere else in the ESSENCE′ model being constructed. When, for example, refining *ring*$_1$ with SIZEDSETEQUALITY1, S'_1 = `genSymbol`(*ring*$_1$, `matrix` [`indexed by` *Nodes*] `of` $1..n$), and so S'_1 will denote a unique identifier for a matrix of decision variables (since *ring*$_1$ is a decision variable).

Definitions via `genSymbol` and type information (except for the input expression) are given on the right of the long vertical bar. The bar itself has no meaning beyond separating the details of the rule from the types and definitions.

Since SIZEDSETEQUALITY1 chooses to represent both sets explicitly, the expression e =“$(\forall i \exists j S'_1[i] = S'_2[j]) \wedge (n_1 = n_2)$” is used to constrain the two explicit matrices to represent the same set (note the introduction of the quantified variables denoted by i and j). Since, in general, it may be the case that e is not yet in ESSENCE′, it is refined further and the result, denoted by ϕ, is returned. In this example, e = “$\forall i\ \exists j\ ring'_1[i] = ring'_2[j]\ \wedge\ (capacity = capacity)$” is in ESSENCE′, so $\rho(e) = e$. For brevity, the model shown in Fig. 3 omits $capacity = capacity$; a simplifier could easily remove it. This convention is followed in all subsequent models generated by SIZEDSETEQUALITY1.

Recall that the explicit representation of a `set(size` n`)` `of` τ is a matrix of n *distinct* elements, each of type τ. Thus, when a rule generates this explicit representation, it must introduce into the generated model a constraint that the elements of the matrix are all different. In particular, the SIZEDSETEQUALITY1 rule must introduce two such constraints: AllDifferent(S'_1) and AllDifferent(S'_2). And, since the elements of S'_1 and S'_2 have not been refined, ρ must be applied to these two constraints, resulting in the ESSENCE′ constraints named χ and ψ in the rule.

Consider where χ and ψ should occur in the generated model. From the ESSENCE′ model of Fig. 3 it might appear that χ and ψ should be conjoined to ϕ, returning $\phi \wedge \chi \wedge \psi$. To see that this is incorrect, observe that this treatment would refine $\neg(ring_1 = ring_2)$ to $\neg(\phi \wedge \chi \wedge \psi)$, whereas the desired refinement is $(\neg\phi) \wedge \chi \wedge \psi$. The correct refinement is obtained by returning ϕ as the refinement, tagging the refinement with $\chi \wedge \psi$, and once the ESSENCE′ model is generated, adding $\chi \wedge \psi$ to its constraints. Since, in this case, the tag is a constraint, it is labelled by `such that`.

When a rule builds a refinement R, a part P of which has been generated from a recursive call to ρ, the tags of P are, by default, added to the tags of R. Hence, in SIZEDSETEQUALITY1, the tags attached to a refinement of ϕ, χ and ψ are implicitly added to the `such that` tag explicitly given in the rule. As will be seen later, this default can be overridden.

Space precludes giving the full derivation of the explicit ESSENCE′ model (Fig. 3). The refinements triggered via SIZEDSETEQUALITY1 in the example are, however, straightforward. Universal quantification over a finite range of integers is in ESSENCE′ and can be viewed as a conjunction or simple ‘for’ loop. Similarly, existential quantification over a finite range of integers can be treated as disjunction. Hence, the first constraint in the explicit ESSENCE′ model comes directly from SIZEDSETEQUALITY1. The occurrence model follows similarly.

After refining the constraints and the objective function of an ESSENCE specification, the `given` `find` and `letting` statements are generated. This is done by scanning the set of constraints and adding an appropriate definition for each unique identifier. `where` statements are refined in the same

SizedSetEquality1 $\rho(S_1 : \texttt{set}\,(\texttt{size}\ n_1)\ \texttt{of}\ \tau = S_2 : \texttt{set}\,(\texttt{size}\ n_2)\ \texttt{of}\ \tau) \overset{\text{ref}}{\rightarrow}$

$\{\quad \phi$
`such that` $\chi \wedge \psi$
$|$
$\phi \in \rho((\forall i \exists j S'_1[i] = S'_2[j]) \wedge (n_1 = n_2))$
$\chi \in \rho(\text{AllDifferent}(S'_1))$
$\psi \in \rho(\text{AllDifferent}(S'_2)) \quad \}$

$S'_1 = \texttt{genSymbol}(S_1, \texttt{matrix}\,[\texttt{indexed by}\ 1..n_1]\ \texttt{of}\ \tau)$
$S'_2 = \texttt{genSymbol}(S_2, \texttt{matrix}\,[\texttt{indexed by}\ 1..n_2]\ \texttt{of}\ \tau)$
$i = \texttt{genSymbol}(\text{“Quantified variable”}, 1..n_1)$
$j = \texttt{genSymbol}(\text{“Quantified variable”}, 1..n_2)$

SizedSetEquality2 $\rho(S_1 : \texttt{set}\,(\texttt{size}\ n_1)\ \texttt{of}\ \tau = S_2 : \texttt{set}\,(\texttt{size}\ n_2)\ \texttt{of}\ \tau) \overset{\text{ref}}{\rightarrow}$

$\{\quad \phi$
`such that` $\chi \wedge \psi$
$|$
$\phi \in \rho((\forall i S'_1[i] = S'_2[i]) \wedge (n_1 = n_2))$
$\chi \in \rho(\Sigma_i S'_1[i] = n_1)$
$\psi \in \rho(\Sigma_i S'_2[i] = n_2) \quad \}$

τ is $[a..b]$ or `bool`
$S'_1 = \texttt{genSymbol}(S_1, \texttt{matrix}\,[\texttt{indexed by}\ \tau]\ \texttt{of bool})$
$S'_2 = \texttt{genSymbol}(S_2, \texttt{matrix}\,[\texttt{indexed by}\ \tau]\ \texttt{of bool})$
$i = \texttt{genSymbol}(\text{“Quantified variable”}, \tau)$

ForallRange1 $\rho(\forall i : n_1..n_2\ \ \phi : \texttt{bool}) \overset{\text{ref}}{\rightarrow}$

$\{\quad$ forall $(i : n_1..n_2)\ \phi'$
`such that` forall $(i : n_1..n_2)\ \Gamma$
$|$
$(\phi'$ `with such that` $\Gamma) \in \rho(\phi) \quad \}$

ForallSizedSet1 $\rho(\forall i : \tau \in S : \texttt{set}\,(\texttt{size}\ n)\ \texttt{of}\ \tau.\phi : \texttt{bool}) \overset{\text{ref}}{\rightarrow}$

$\{\quad$ forall $(j : 1..n)\ \phi'$
`such that` $\chi \wedge$ forall $(j : 1..n)\ \Gamma$
$|$
$(\phi'$ `with such that` $\Gamma) \in \rho(\phi[i \mapsto S'[j]])$
$\chi \in \rho(\text{AllDifferent}(S')) \quad \}$

$j = \texttt{genSymbol}(i, 1..n)$
$S' = \texttt{genSymbol}(S, \texttt{matrix}\,[\texttt{indexed by}\ 1..n]\ \texttt{of}\ \tau)$

SizedSubset1 $\rho(S_1 : \texttt{set}\,(\texttt{size}\ n_1)\ \texttt{of}\ \tau \subseteq S_2 : \texttt{set}\,(\texttt{size}\ n_2)\ \texttt{of}\ \tau) \overset{\text{ref}}{\rightarrow}$

$\{\quad \phi$
`such that` $\chi \wedge \psi$
$|$
$\phi \in \rho(\forall i . S'_2[S'_1[i]])$
$\chi \in \rho(\Sigma_j S'_2[j] = n_2)$
$\psi \in \rho(\text{AllDifferent}(S'_1)) \quad \}$

τ is $[a..b]$ or `bool`
$S'_1 = \texttt{genSymbol}(S_1, \texttt{matrix}\,[\texttt{indexed by}\ 1..n_1]\ \texttt{of}\ \tau)$
$S'_2 = \texttt{genSymbol}(S_2, \texttt{matrix}\,[\texttt{indexed by}\ \tau]\ \texttt{of bool})$
$i = \texttt{genSymbol}(\text{“Quantified variable”}, 1..n_1)$
$j = \texttt{genSymbol}(\text{“Quantified variable”}, \tau)$

Figure 4: Example Refinement rules.

way that constraints are.

Refining Nested Types: A Simple Example

To see how expressions of a nested type are refined, consider the Microsonet$_2$ specification (Fig. 5), where the goal is to generate two identical sets of rings. The key decision is the representation of *rings*$_1$ and *rings*$_2$, two sets of sets. Refinement begins with $\rho($*rings*$_1$ = *rings*$_2)$, requiring SIZEDSETEQUALITY. Of the rules given in Fig. 4, only SIZEDSETEQUALITY1 is applicable, since “`set (size` *capacity*`) of` *Nodes*” is neither a range of integers nor Boolean: indexing a matrix by a complex type (here each index would correspond to one of the possible sets of *Nodes*) would often lead to unfeasibly large matrices. Future work will consider relaxing this condition in certain cases.

The justification for refining the arguments of the equality constraint inside the equality rule itself becomes clear in the context of nested types. A seemingly more-natural approach to refining $S_1 = S_2$ would be to refine S_1 and S_2 and constrain the results to be equal. However, given unbounded nesting of types and multiple possible refinements of each type, constraining the results to be equal is not straightforward. Consider the case where S_1 and S_2 are not sets of integers, but sets of sets of . . . sets of integers. Having refined S_1 and S_2 there is no fixed constraint on their refinements that enforces equality on S_1 and S_2. Instead of this approach, our rules are designed to “peel off” a layer of nesting so that the associated constraint/operator can be formed without looking arbitrarily deep inside a refined expression.

Performing SIZEDSETEQUALITY1 requires refining the two `such that` tags, both of which impose that all the elements of a matrix of sets must be different. Since these elements may be of a non-atomic type, this AllDifferent constraint must be refined. For space reasons the definition of the ALLDIFFERENT refinement rules are omitted.

The main constraint generated by SIZEDSETEQUALITY1 is $\rho(\forall i \exists j\, rings'_1[i] = rings'_2[j])$. The FORALLRANGE equation is now applicable (Fig. 4 gives FORALLRANGE1). This rule works by refining ϕ, which contains the free variable i. Each resulting refinement ϕ' and its tag Γ, both of which contain the free variable i, are wrapped in “forall $i : n_1..n_2$.” Refinement continues with $\rho(\exists j\ \ rings'_1[i] = rings'_2[j])$. EXISTSRANGE operates in the same way as FORALLRANGE, so we omit the details.

We now have $\rho(rings'_1[i] = rings'_2[j])$, for which SIZEDSETEQUALITY is used. Both rules are applicable, since $rings'_1[i]$ and $rings'_2[i]$ are sets of *Nodes*. The refinement in Fig. 5 uses SIZEDSETEQUALITY2 to generate an occurrence representation of the inner sets. The rule operates as described earlier, but we highlight how `genSymbol` treats indexed matrices. `genSymbol` respects the index structure of its first argument. Hence, given the indexed one-dimensional array $rings'_1[i]$, `genSymbol`$(rings'_1[i],$ `matrix`...$)$ creates a two-dimensional Boolean matrix, $rings''_1$ (see Fig.5), and returns the partially-indexed $rings''[i]$, which is the refined version of an element of the set of sets.

The remaining refinements are straightforward. Note that $rings'_1$, $rings'_2$ are not in the model. These are intermediate representations which, although more concrete than *rings*$_1$

```
given        nrings : int, nnodes : int, capacity : int
where        nrings ≥ 1, nnodes ≥ 1, capacity ≥ 1
letting      Nodes be 1..nnodes
find         rings_1 : set (size nrings) of set (size capacity) of Nodes
             rings_2 : set (size nrings) of set (size capacity) of Nodes
such that    rings_1 = rings_2
```

```
given       nrings : int, nnodes : int, capacity : int
where       nrings ≥ 1, nnodes ≥ 1, capacity ≥ 1
letting     Nodes be 1..nnodes
find        rings_1'' : matrix [indexed by 1..nrings, Nodes] of bool
            rings_2'' : matrix [indexed by 1..nrings, Nodes] of bool
such that   forall (i : 1..nrings) exists (j : 1..nrings)
               forall (k : Nodes) rings_1''[i][k] = rings_2''[j][k]
            forall (i : 1..nrings)  sum (j : Nodes)  (rings_1''[i][j]) = capacity
            forall (i : 1..nrings)  sum (j : Nodes)  (rings_2''[i][j]) = capacity
            forall (i, j : 1..nrings) i < j implies
               exists (k : Nodes) rings_1''[i][k] ≠ rings_1''[j][k]
            forall (i, j : 1..nrings) i < j implies
               exists (k : Nodes) rings_2''[i][k] ≠ rings_2''[j][k]
```

Figure 5: ESSENCE specification and an ESSENCE′ model of the MicroSonet$_2$ problem.

```
given        nrings : int, nnodes : int, capacity : int
where        nrings ≥ 1, nnodes ≥ 1, capacity ≥ 1
letting      Nodes be 1..nnodes
given        demand : set (size m) of set (size 2) of Nodes
find         rings : set (size nrings) of set (size capacity) of Nodes
such that    ∀pair ∈ demand. ∃r ∈ rings . pair ⊆ r
```

```
given       nrings : int, nnodes : int, capacity : int
where       nrings ≥ 1, nnodes ≥ 1, capacity ≥ 1
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
such that   forall (i : 1..m)  exists (j : 1..nrings)  forall (k : 1..2)
               rings''[j][demand''[i][k]]
            forall (i : 1..nrings)  sum (j : Nodes)  (rings''[i][j]) = capacity
            forall (i, j : 1..nrings) i < j implies
               exists (k : Nodes) rings''[i][k] ≠ rings''[j][k]
```

Figure 6: ESSENCE specification and an ESSENCE′ model of the MicroSonet$_3$ problem.

and *rings*$_2$, are matrices of sets, and so not in ESSENCE′.

Refining Nested Types: A Complex Example

Our final example illustrates a difficult case when refining nested types: quantification over nested types. Consider MicroSonet$_3$ (Fig. 6), a variant of a large piece of SONET, where the goal is to fill a given number of rings such that the inter-node communication demand is met. Refinement begins with $\rho(\forall pair \in demand. \exists r \in rings.pair \subseteq r)$, using the FORALLSIZEDSET1 rule (Fig. 4). The first step is to refine the binding expression, *pair* $\in$ *demand*, which is, in turn, dependent on the refinement of the given set *demand*. FORALLSIZEDSET1 represents *demand* explicitly in the same way as SIZEDSETEQUALITY1. The returned expression quantifies over the indices of this matrix.

The quantified expression ϕ (which is $\exists r \in rings.pair \subseteq r$ in the example) is refined in the context of the decision to represent the set S explicitly as the matrix S'. In doing so, $S'[j]$ is substituted for i via $\phi[i \mapsto S'[j]]$ (in the example $demand'[j]$ is substituted for *pair*). As per FORALLRANGE1, the tags associated with this refinement must also be quantified to hold for each possible value of j. Having highlighted this case, space precludes giving further details of the refinement. One particular rule, SIZEDSUBSET1, which is central to this specification, is presented in Fig. 4. This rule shows two expressions, S_1 and S_2, of the same type being refined to two different representations. The design of CONJURE means this is no more complicated than the previously presented rules, which produce identical representations.

Refining Nested Operators

None of the refinement examples considered so far have used an ESSENCE specification with nested operators. We have avoided the extremely difficult task of formulating refinement rules that recur on both the type of an expression and its syntactic structure. Each recursive rule either works recursively on syntactic structure and handles only atomic types or it works recursively on nested types and handles only expressions that are syntactically "primitive" (in a sense not defined here). A constraint containing *both* nested types and non-primitive syntactic structure is refined by first rewriting it to an equivalent one containing only primitive syntactic structure. For example, the constraint $A \cap (B \cup C) \subseteq D$ is rewritten to $\exists X, Y \ (X = B \cup C) \wedge (Y = A \cap X) \wedge (X \subseteq D)$. Each conjunct of the resulting constraint is a primitive constraint that has a dedicated refinement rule. CONJURE performs this rewriting in a preprocessing stage called *flattening*.

5 Some Symmetry Detection is Free

Refinement often introduces symmetries into the models it generates; that is, the generated model contains symmetries that do not correspond to symmetries in the original specification. This always happens when a variable X in the specification is represented by a collection of variables $X'_1, \ldots, X'_n$ in the model such that an instantiation of X corresponds to multiple instantiations of $X'_1, \ldots, X'_n$. For example, if X is of type `set (size 3)`, its explicit representation is a matrix of three variables, $X'[1..3]$. An instantiation of X corresponds to 3! instantiations of $X'[1..3]$. The matrix X' has index symmetry [6], which can broken by the symmetry-breaking constraint $X'[1] \leq X'[2] \leq X'[3]$.

Since certain rules always introduce symmetry, such a rule can tag the refined expression with a description of the symmetry. For example, (though not shown in Fig. 4) SIZEDSETEQUALITY1 tags ϕ with "`symmetry` *indexsym*(ϕ, 1)," which indicates that ϕ has symmetry in its first index. Once a model has been generated, the `symmetry` tags can be used to add appropriate symmetry-breaking constraints, or to guide a symmetry-aware search method such as SBDS or SBDD.

Detecting symmetry as refinement introduces it is a considerably cheaper operation than symmetry detection on a finished model. In future, we intend to extend our system of `symmetry` tags to include other types of symmetry, such as partial index symmetry and value symmetry.

6 Generating Models with Channelling

As we have seen, ESSENCE expressions have multiple possible refinements. If the same expression appears multiple times in a specification, then, different refinements of it may appear in an ESSENCE′ model. Sometimes these concrete representations have a different name but they are equivalent. In those cases CONJURE automatically unifies them into a single representation. However, CONJURE often produces distinct concrete representations for the same abstract symbol. At first sight, this may appear to be a weakness. On the contrary, some of the most effective models in the literature take this approach, since some representations allow certain constraints to be modelled more easily than others. To maintain consistency among the different representations, *channelling constraints* are used [4].

To facilitate channelling, (though not shown in Fig 4) the refinement rules add `represent` tags to the refinements to record the relationship between ESSENCE identifiers and their representations in the concrete model. For example, when refining a decision variable *S* of type `set (size` n`) of int` it is possible for *S* to be represented by both occurrence and explicit representations in the same model. If that is the case the model will contain the tags "`represent` *S* by *expset*(S_1)" and "`represent` *S* by *occset*(S_2)," connecting *S* with its concrete representations S_1 and S_2. By following the chain of tags it is possible to connect each concrete object to the original entity it is representing, and from these it is possible to construct channelling constraints.

7 Experimental Results

This paper makes an empirically testable conjecture: it is possible to develop good constraint models for combinatorial problems by specifying the problem in ESSENCE and refining this specification with CONJURE. At this point we can test the conjecture up to certain limits. The first limitation is that we currently are concerned only with the kernel of the model, which lacks channelling constraints and symmetry-breaking constraints and which has not had any transformations applied. The second limitation is that the implementation of CONJURE currently has rules for only certain types: integers, integer ranges, Booleans, bounded and fixed-size sets, bounded and fixed-size multisets, and all compositions of these. The implemented rules can generate ESSENCE$'$ models with occurrence and explicit representations and with variables whose domains contain sets of integers.

We test this conjecture by identifying problems that can be specified using only the supported types, and checking whether CONJURE generates the kernel of all reasonable models, particularly models appearing in the literature. Most problems in the literature cannot be specified abstractly with the supported types since they involve associating together elements from two or more sets, which requires types such as functions or relations. We have found seven problems in the literature that can be specified abstractly with only the supported types: Schur's Lemma (SL), Balanced Incomplete Block Design (BIBD), Steiner Triple Systems (STS), SONET (SONET), Social Golfers Problem (SGP), RL Design (RLD), and Balance Code Generation (BCG). Though we know of no published models for RLD and BCG (both come from [5]), we have included them simply to test whether CONJURE can generate reasonable, correct models.

The results of running CONJURE on ESSENCE specifications of these seven problems are summarised in the Table 1.

Problem	#	Choices	Yes	No
BIBD	9	dv: mset of sets (2)	[12]	-
SL	9	dv: mset of sets (2)	[6]	[6]
RLD	9	dv: mset of sets (2)	-	-
STS	27	dv: mset of sets (3)	[12][13]	-
BCG	3	dv: mset of sets (1)	-	-
SONET	27	dv: mset of sets (2) in: set of sets (1)	4×[8]	4×[8]
SGP	54	dv: mset of set of sets (2)	[12][13]	[16]

Table 1: Results of running CONJURE on seven problems.

To see how to interpret the table, consider the row labelled SONET. This indicates that CONJURE generated 27 models for this problem. The choices it had to make were how to represent each of the two occurrences of the decision variable (whose type is set of set) and the one occurrence of the input parameter (whose type is set of sets). Our survey of the literature uncovered four published models (all in [8]) that were among those generated by CONJURE and four (all in [8]) that were not. The entry for SGP indicates that CONJURE generated two of the three models that we found in the literature. Though some models appear in multiple published papers, the table cites only one.

Examining the results of the experiments, we observed that all of the models generated by CONJURE were correct. The kernels of many models found in the literature were generated by CONJURE; that is, the models use the same decision variables and the same problem constraints (as opposed to implied or channelling constraints).

It is more enlightening to consider the models not generated by CONJURE. The four models of SONET not generated by CONJURE arise straightforwardly by considering the decision variable to be a relation between rings and nodes. The rules for refining a relation in this way could be expressed easily in the framework of CONJURE and would enable the generation of these four additional models. This illustrates that there are important issues involved in writing specifications in ESSENCE, which in this case are also intertwined with the detection of symmetry; all of this is outside the scope of this paper.

The missing model of SL could be generated easily from the most natural expression of the problem, which takes the decision variable to be a partition. Again, it would be straightforward to add to CONJURE the necessary rules for partitions.

Finally, the model of SGP not generated by CONJURE uses a novel modelling technique that was created through insight into the structure of the problem. We have not yet considered whether that technique can be generalised and then captured in CONJURE rules.

8 Related Work in Modelling Support

OPL [19], a modelling language similar to ESSENCE$'$, has been extended by ESRA [7] to provide relation variables and by $\mathcal{F}$ [11] to provide function variables. ESRA currently compiles to only a single representation, but $\mathcal{F}$ can produce alternative models, some with multiple representations and channelling. Other work has extended constraint languages to support a single representation of both sets [10] and multisets [20]. However our work is unique in providing nested types.

Other languages have been used to specify combinatorial problems. NP-SPEC [3] uses existential second-order logic to express executable specifications of every NP-complete problem. Renker et al. [15] discuss representing abstract specifications in Z, an expressive abstract language that is supported by automatic type-checkers and interactive tools. However, refinement and transformation of the Z specification is manual.

Multi-tac [14] and CACP [2; 1] are systems that perform small-scale transformations of constraint specifications and use heuristics and search to try to improve the search strategy used in the solving of CSPs. These and similar systems are important steps towards automating the solving of CSPs, and complement our work on CONJURE, which at present is only concerned with refining of high-level specifications and not the search strategy to be used, and does not implement transformation rules as of yet.

The area of automatic program refinement is a long-standing and mature area of computer science. KIDS [17] and Specware [18] are among the many domain-independent systems that support the interactive or automatic refinement of formal specifications into *imperative* programs. Our work on CONJURE operates in a simpler setting of transforming specifications into finite-domain constraint models, which are *declarative*. Because our setting is simpler, we can be more ambitious about the extent of automation achieved.

9 Conclusion

This paper has made two major contributions towards realising the possibility of formalising and automating the generation of constraint models. First, the paper has presented a language that enables combinatorial problems to be specified at a level above that at which modelling decisions take place. Second, the paper has shown how a system of formal rules could refine such a specification into model kernels at the level of abstraction supported by existing constraint solvers. We have also outlined an approach to integrating other aspects of model generation—channelling and symmetry breaking—into the existing system.

A rigorous account of model generation would be valuable in several ways. It could 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 in constructing a catalogue of modelling constructs.

The automation of model generation offers further opportunities. CONJURE currently uses its rules to generate all possible models; but the rules could also be used within an interactive system, which, like an interactive theorem prover, allows the user to chose from among the alternatives available at a decision point. Our ultimate dream is that the automation of model generation takes us one step closer to automating the entire modelling process.

Acknowledgements Ian Miguel is supported by a UK-Royal Academy of Engineering/EPSRC Postdoctoral Fellowship. We thank Adam Bakewell for contributing to an early version of this work, and Brahim Hnich and Toby Walsh for useful discussions.

References

[1] J. Borrett and E.P.K. Tsang. A context for constraint satisfaction problem formulation. *Constraints*, 6:299–327, 2001.

[2] R. Bradwell, R. Ford, J. Mill, E.P.K. Tsang, and R. Williams. An overview of the CACP project: Modelling and solving constraint satisfaction/optimisation problems with minimal expert intervention. *Wkshp. on Analysis & Visualization of Constraint Programs & Solvers*, 2000.

[3] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.

[4] B. M. W. Cheng, K. M. F. Choi, J. Ho-Man Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: An experience report. *Constraints*, 4:167–192, 1999.

[5] C. J. Colbourn and J. H. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC, 1995.

[6] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. *Proc. 8th Int. Conf. on Princ. & Pract. of CP, LNCS* 2470, 462–476, 2002.

[7] P. Flener, J. Pearson, M. Agren Introducing ESRA, a relational language for modelling combinatorial problems. LOPSTR'03: Revised Selected Papers, LNCS 3018, 214-232, 2004.

[8] A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, and T. Walsh. Transforming and refining abstract constraint specifications. *Proc. of CSCLP04: Joint Annual Workshop of ERCIM/Colognet on Constraint Solving & Constraint Logic Programming*, 74–88, 2004.

[9] I. P. Gent and T. Walsh. CSPLib: A benchmark library for constraints. Technical Report APES-09-1999, APES, 1999.

[10] C. Gervet. Conjunto: Constraint logic programming with finite set domains. *Proc. Int. Symposium in Logic Programming*, 339–358. The MIT press, 1994.

[11] B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, Dept. Information Science, Uppsala Univ, 2003.

[12] Z. Kiziltan. *Symmetry Breaking Ordering Constraints*. PhD thesis, Dept. Information Science, Uppsala Univ., 2004.

[13] V. Lagoon and P. J. Stuckey. Set domain propagation using ROBDDs. *Proc. 10th Int. Conf. on Principles & Practice of Constraint Programming*, 347–361, 2004.

[14] S. Minton. Integrating heuristics for constraint satisfaction problems: A case study. *Nat. Conf. on AI*, 120–126, 1993.

[15] G. Renker and A. Ahriz. Building models through formal specification. *Proc. CP-AI-OR-04*, LNCS 3011, 295–401, 2004.

[16] B. M. Smith. Reducing symmetry in a combinatorial design problem. *Proc. CP-AI-OR*, 351–359, 2001.

[17] D. R. Smith. KIDS: A knowledge-based software development system. *Automating Software Design*, 483–514. MIT Press, 1991.

[18] *Specware 4.0 User Manual*. Kestrel Institute, October 1994.

[19] P. van Hentenryck. The OPL Optimization Programming Language. MIT Press, 1999.

[20] T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. *Proc. 9th Int. Conf. on Princ. & Pract. of CP*, LNCS 2833, 724–738, 2003.