Towards Automatic Modelling of Constraint Satisfaction Problems: A System Based on Compositional Refinement

Adam Bakewell, Alan M. Frisch, and Ian Miguel

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

Abstract. Many and diverse search problems have been solved with great success using constraint programming. However, in order to apply constraint programming tools to a particular domain, the problem 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 on the part of the modeller. This paper describes a prototype automated modelling 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 high-level specification language 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

Modelling combinatorial problems in a constraint programming language can be automated. In particular, the task of transforming a combinatorial problem in an abstract, high-level specification language into an effective program (specified in an existing constraint programming language) for solving the problem can be automated. This is our conjecture and it is the aim of this paper to provide initial support for it.

The successful development of an automated modelling system would have significant practical consequences. To solve complex—and, often, important—problems with existing constraint satisfaction toolkits requires considerable expertise in modelling the problem with the low-level constructs provided by the toolkit. Solving the problem with the use of an automated modelling system would require significantly less expertise; one would need only to specify the problem in a high-level language. By bringing this language closer to the way that humans conceptualise problems and express them in natural language, we reduce the expertise required for specifying the problem.

Our belief is that modelling expertise can be systematised, catalogued and, ultimately, formalised and embedded in an automated system. Not only would

the automated system be useful, but a catalogue of systematic modelling techniques would be of great benefit to the constraint modelling community. Such a catalogue could be consulted to find appropriate modelling techniques and to find what is known about the tradeoffs among alternatives. This idea is detailed and illustrated under the label of "constraint patterns" [9].

We speculate that in some respects an automated modelling system could outperform a human expert. The automated system could embody the modelling techniques that are employed by many experts, thus having at its disposal more techniques than any single expert. The automated system could be systematic in considering combinations of alternatives, whereas a human might overlook some. Also, a human might be overwhelmed by the size and complexity of a specification. We believe that a system could be highly-capable of combining existing modelling techniques in novel and complex ways. However, it is not our goal to produce a system that could discover new modelling techniques.

We divide our pursuit of producing an automated modelling system into two stages. The goal of the first stage, the *competence stage*, is to develop a system that generates a set of feasible models for a given specification. This set should contain all the effective models that would be produced by a human expert modeller. The set might also contain ineffective models and effective models that might not be produced by a human expert. The goal of the second stage of research, the *performance stage*, is to endow the system with some ability to evaluate the models it generates. Thus, the system would not generate models that are obviously ineffective and, as human experts can do, it might suggest that some of the alternatives are likely to be more effective than others.

This paper contributes to the goals of the competence stage. We believe that the key to the competence stage is a fundamental conjecture, *model compositionality*: the effective models for a combinatorial problem can be constructed by composing models for each component of the problem specification.[1] The contribution of this paper is to demonstrate, by example, that refinement can be performed compositionally; that is, a set of alternative refinements of a specification can be generated by composing refinements of the components of the specification. In particular, we present a set of compositional refinement rules that can transform a specification in a high-level source language into alternative models expressed in a target language whose constructs are similar to those provided by current constraint programming languages. The alternative models employ different structures for representing the decision variables and some employ redundant structures linked by channelling constraints, which are also generated automatically.

We have implemented in Haskell an automated refinement system that incorporates a set of refinement rules that generalises and extends those presented here as well as a β-reduction rule to simplify λ-expressions. The tool automatically produces models for problems like BIBD—for which it generates 8 models—and the SONET problem [7] for which it generates 64 models.

[1] Of course, the ways that models compose may be highly complex and uncovering them may be difficult.

2 A Brief Introduction to Our Specification Language

This section briefly introduces the specification language in which we specify problems to be refined. Since this paper focuses on refinement rather than language design, we present only those features of the language that are necessary to proceed with the presentation of refinement. Here we merely present the language; later we discuss how the demands of refinement have driven us to this design.

Throughout this and the following two sections, the *Balanced Incomplete Block Design* (BIBD) problem (problem 28 at `www.csplib.org`) is used as an example. Despite its relative simplicity, this problem has important practical applications, such as cryptography and experimental design. Here it illustrates our language. The following is an English definition of the BIBD problem (from CSPLib).

> Given positive integers, nv, nb, k, r and l, arrange nv distinct objects in nb blocks such that each block contains k distinct objects, each object occurs in exactly r different blocks and every two distinct objects occur together in exactly l blocks.

With this problem in mind, we now turn our attention to our specification language. A problem specification is a triple consisting of a list of *declarations*, a set of *decision variables* and a set of *constraints*. We write this triple as

declarations `find` *decision variables* `such that` *constraints*

The list of declarations is used to define *parameters* and *constants*. The parameters to a problem specify the instance of the problem class. The value of a parameter is provided by the user in specifying the problem instance to be solved. The BIBD problem has parameters nv, nb, r, k, and l. The decision variables denote the combinatorial objects that are to be found by the solver. The goal in the BIBD problem is to find a relation, which we shall call $BIBD$, between the blocks and the objects.

The specification language is strongly typed and every expression has a type. The five parameters in the BIBD problem are of type *natural*. The type of relation $BIBD$ is the powerset of $B \times V$, written $\wp(B \times V)$, where B and V are primitive types representing the blocks and objects. B and V are constant types and therefore must be declared in the list of declarations. Putting these pieces together, the specification of the BIBD is of the form

`given` nv, nb, r, k, l : nat
`letting` B be type-of-size nb, V be type-of-size nv
`find` $BIBD : \wp(B \times V)$
`such that` *constraints*

We see here that *declarations* is a list of intermingled `given` statements, which declare the parameters, and `letting` statements, which declare the constants (which may themselves be parameterised). B and V are declared to be

primitive types. As in many constraint specification languages, our language allows a type (or domain) to be specified by enumerating the members. Unlike other languages, a type can also be specified by giving its size and not enumerating its members.

The specification language supports 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 $\tau_1, \ldots, \tau_n$ are types then so are $\tau_1 \times \cdots \times \tau_n$ (a cross product), $\wp(\tau_1)$ (the powerset of τ_1), $\wp_n(\tau_1)$ (the set of all n-element subsets of τ_1), $\tau_1 \rightarrow \tau_2$ (a partial function from τ_1 to τ_2), partition τ_1 (a partitioning of τ_1), and regpart $\tau_1\, n$ (a partitioning of τ_1 into subsets of size n).

Constraints in this specification language are built from the declared parameters, constants and decision variables using operators commonly found in mathematics and in other constraint specification languages. The language also includes universally quantified variables ranging over any specified finite type. Each subexpression in a constraint is typed. The *constraints* component of the BIBD problem can be expressed as

$$\forall b : B \cdot |BIBD(b, \_)| = k \tag{1}$$

$$\forall v : V \cdot |BIBD(\_, v)| = r \tag{2}$$

$$\forall \{v, v'\} : \wp(V) \cdot |BIBD(\_, v) \cap BIBD(\_, v')| = l \tag{3}$$

Here $\{v, v'\} : \wp(V)$ is a two-element set containing objects of type V, and $BIBD(b, \_)$ is the projection of $BIBD$ onto b—that is, $\{v : V | BIBD(b, v)\}$.

3 Introducing Refinement Rules

This section introduces a set of *refinement rules* that refine specifications into constraint programs. Each rule specifies, by means of a refinement operator ρ, the refinement of a specification in our specification language into a constraint language similar to a constraint program. Refinement requires a number of mechanisms, which are introduced incrementally through a sequence of rule sets of increasing power. The rule sets are distinguished by subscripting the ρ operator.

The refinement rules in our implementation are formulated to handle relations of arbitrary arity. To simplify the presentation, the rules presented in this paper operate on relations of only a fixed arity.

3.1 Building One Model at a Time

Let us now present domain independent refinement rules using the refinement of the BIBD problem as an example. The rule sets given in this sub-section are simplified in that they assume that a single model is being produced using one type of target-level object. The subsequent sub-section considers how to produce a set of alternative models containing different types of object.

Refinement to a 01 Matrix Model A binary relation variable $R : \wp(D_1 \times D_2)$ can be represented by a 2-dimensional 01 matrix that is indexed by D_1 and D_2, which is named R_{M01}.

RELATION1: $\rho_1(R : \wp(D_1 \times D_2)) = \lambda d_1, d_2 . R_{M01}[d_1, d_2] : D_1 \times D_2 \to 01$

The relationship between the two structures is:

$$\forall d_1 : D_1, d_2 : D_2 \cdot R(d_1, d_2) \leftrightarrow R_{M01}[d_1, d_2] = 1 \tag{4}$$

We have two rules to refine a projection onto either argument of a binary relation. For any expression ϕ of type $\wp(D_1 \times D_2)$, any x of type D_1 and any x' of type D_2:

PROJECTION1A: $\rho_1(\phi(x, \_)) = \lambda d_2 . (\rho_1(\phi)(x, d_2)) : D_2 \to 01$
PROJECTION1B: $\rho_1(\phi(\_, x')) = \lambda d_1 . (\rho_1(\phi)(d_1, x')) : D_1 \to 01$

The Application of the RELATION1 and PROJECTION1A rules to $BIBD(b, \_)$,a sub-expression of constraint (1), produces:

$$\rho_1(BIBD(b, \_)) = \lambda v . BIBD_{M01}[b, v]$$

Another rule refines the cardinality of a relational expression ϕ:

CARDINALITY1: $\rho_1(|\phi|) = \sum_{d:D} \rho_1(\phi)(d) : \text{nat}$, where $D \to 01$ is the type of $\rho_1(\phi)$

Three more rules are required to complete the example refinement:

EQUALITY1: $\rho_1(\phi = \phi') = (\rho_1(\phi) = \rho_1(\phi')) : 01$.
CONST/PARAM1: $\rho_1(k : D) = k : D$, where k is any constant or parameter or quantified variable.
QUANTIFIER1: $\rho_1(\forall d : D \cdot \phi) = \forall d : D \cdot \rho_1(\phi)$

Using the given rules, constraints (1) and (2) can now be refined:

$$\rho_1(\forall b : B \cdot |BIBD(b, \_)| = k) = \forall b : B \cdot \sum_{v:V} BIBD_{M01}[b, v] = k$$
$$\rho_1(\forall v : V \cdot |BIBD(\_, v)| = r) = \forall v : V \cdot \sum_{b:B} BIBD_{M01}[b, v] = r$$

To refine constraint (3), we need one further rule for the intersection of two expressions ϕ and ϕ', both of type $\wp(D)$.

INTERSECTION1: $\rho_1(\phi \cap \phi') = (\lambda d . (\rho_1(\phi)(d) \wedge \rho_1(\phi')(d))) : D \to 01$

The refinement of constraint (3) completes the refinement of the specification of the BIBD problem to a 01 matrix model:

$$\rho_1(\forall \{v, v'\} : \wp(V) \cdot |BIBD(\_, v) \cap BIBD(\_, v')| = l) =$$
$$\forall \{v, v'\} : \wp(V) \cdot \sum_{b \in B} BIBD_{M01}[b, v] \wedge BIBD_{M01}[b, v'] = l$$

Refinement to 1-d Matrix of Sets Indexed by First Argument The previous subsection showed how the well-known 01 matrix model of the BIBD problem can be generated by refining the relational description. There are, however, several other alternatives. For example, another way to represent the BIBD relation is with a 1-dimensional matrix of set variables. Here we specify a refinement operator, ρ_2, to generate this alternative.

For every relation $R : \wp(D_1 \times D_2)$, R_{MD_1} denotes a 1-dimensional matrix indexed by D_1 whose elements are subsets of D_2.

RELATION2: $\rho_2(R : \wp(D_1 \times D_2)) = \lambda d.R_{MD_1}[d] : D_1 \rightarrow \wp(D_2)$

The relationship between the two structures is:

$$\forall \langle d_1, d_2 \rangle : D_1 \times D_2 \cdot R(d_1, d_2) \leftrightarrow d_2 \in R_{MD_1}[d_1] \tag{5}$$

The refinement rules for the other constructs are identical to those for ρ_1, except that we replace PROJECTION1A, CARDINALITY1 and INTERSECTION1 with

PROJECTION2A: $\rho_2(\phi(x, \_)) = \rho_2(\phi)(x) : \wp(D_2)$, where $\rho_2(\phi)$ has type $D_1 \rightarrow \wp(D_2)$ and x has type D_1.
CARDINALITY2: $\rho_2(|\phi|) = |\rho_2(\phi)| : \text{nat}$, where $\rho_2(\phi)$ has type $\wp(D)$.
INTERSECTION2: $\rho_2(\phi \cap \phi') = (\rho_2(\phi) \cap \rho_2(\phi')) : D$, where $\rho_2(\phi), \rho_2(\phi')$ have type $\wp(D)$.

In CARDINALITY2 we have assumed that set variables have a built-in cardinality operator. There is no PROJECTION2B rule as there is no direct way to implement it with this model of the $BIBD$ relation.

Constraint (1) can now be refined as follows:

$$\rho_2(\forall b : B \cdot |BIBD(b, \_)| = k) = \forall b : B \cdot |BIBD_{MB}[b]| = k \tag{6}$$

Constraints (2) and (3) cannot be refined because there is no PROJECTION2B to refine a projection onto the second argument.

Refinement to 1-d Matrix of Sets Indexed by Second Argument Of course, we can also model the $BIBD$ relation by a 1-dimensional matrix indexed by the second argument of the relation. Here we specify a refinement operator, ρ_3, to generate this alternative.

For every relation $R : \wp(D_1 \times D_2)$, R_{MD_2} denotes a 1-dimensional matrix indexed by D_2 whose elements are subsets of D_2.

RELATION3: $\rho_3(R : \wp(D_1 \times D_2)) = \lambda d.R_{MD_2}[d] : D_2 \rightarrow \wp(D_1)$

The relationship between the two structures is:

$$\forall \langle d_1, d_2 \rangle : D_1 \times D_2 \cdot R(d_1, d_2) \leftrightarrow d_1 \in R_{MD_2}[d_2] \tag{7}$$

The refinement rules for the other constructs are identical to those for ρ_2, except that we replace PROJECTION2A with

PROJECTION3B: $\rho_3(\phi(\_, x)) = \rho_3(\phi)(x) : \wp(D_1)$, where $\rho_3(\phi)$ has type $\wp(D_1 \times D_2)$, and x has type D_2.

This time there is no PROJECTION3A rule as there is no direct way to implement it with this model of the $BIBD$ relation.

Constraints (2) and (3) can now be refined as follows:

$$\rho_3(\forall v : V \cdot |BIBD(\_, v)| = r) = \forall v : V \cdot |BIBD_{MV}[b]| = r \quad (8)$$

$$\rho_3(\forall \{v, v'\} : \wp(V) \cdot |BIBD(\_, v) \cap BIBD(\_, v')| = l) = \quad (9)$$
$$\forall \{v, v'\} : \wp(V) \cdot |BIBD_{MV}[v] \cap BIBD_{MV}[v']| = l$$

However, constraint (1) cannot be refined into this representation. The solution to this problem is to use ρ_2 to model constraints (1), use ρ_3 to model constraints (2) and (3), and introduce *channelling* constraints [1] to connect the two representations. This necessitates the use of a more powerful rule mechanism that enables a single rule to generate multiple representations simultaneously. We now turn our attention to this.

3.2 Building All Models

Good CP models are often *hybrids* that contain multiple representations of an object of interest. This is beneficial when some constraints are easily stated on one representation, and other constraints are easily stated on an alternative representation. As in the refinement of the BIBD problem in the previous section, constraint (1) might be refined using R_{MB}, while constraints (2) and (3) might be refined using R_{MV}. In order to produce such models, a refinement system must:

1. have rules that support each alternative representation,
2. select, according to type, the correct refinement in the context of a partially refined expression, and
3. add channelling constraints to maintain consistency between different representations.

Point 1 is achieved by defining the ρ_4 operator to produce a *set* of typed refined expressions. For example, a single relation refinement rule can produce the three refinements that were previously generated by the RELATION1, RELATION2 and RELATION3 rules:

RELATION4: $\rho_4(R : \wp(D_1 \times D_2)) = \{ \lambda d_1, d_2 . R_{M01}[d_1, d_2] : D_1 \times D_2 \to 01,$
$\lambda d . R_{MD_1}[d] : D_1 \to \wp(D_2),$
$\lambda d . R_{MD_2}[d] : D_2 \to \wp(D_1) \}$

That is, a binary relation variable is refined to a set of three typed expressions, in this case a 01 matrix and two one-dimensional matrices of sets.

Point 2 is achieved by selecting from the refined expressions by type. For example, consider the following rule for projection of a relation onto its first argument:

PROJECTION4A: $\rho_4(\phi(v, \_)) =$
$\{\lambda d_2.(X(v, d_2)) : D_2 \to 01 | X : D_1 \times D_2 \to 01 \in \rho_4(\phi)\} \quad \cup$
$\{X(v) : \wp(D_2) | X : D_1 \to \wp(D_2) \in \rho_4(\phi)\}$

The set of expressions produced by this rule contains at most one element per element of the set of refinements of ϕ. Each expression is generated according to the type of the corresponding refinement of ϕ, as dictated by the conditions.

As a further example, the following rule refines the intersection of two binary relations:

INTERSECTION4: $\rho_4(\phi \cap \phi') =$
$\{(\lambda d_1, d_2.X(d_1, d_2) \wedge X'(d_1, d_2)) : D_1 \times D_2 \to 01$
$\quad | \ (X : D_1 \times D_2 \to 01 \in \rho_4(\phi)) \wedge (X' : D_1 \times D_2 \to 01 \in \rho_4(\phi'))\} \quad \cup$
$\{(\lambda d_1.X(d_1) \cap X'(d_1)) : D_1 \to \wp(D_2)$
$\quad | \ (X : D_1 \to \wp(D_2) \in \rho_4(\phi)) \wedge (X' : D_1 \to \wp(D_2) \in \rho_4(\phi'))\} \quad \cup$
$\{(\lambda d_2.X(d_2) \cap X'(d_2)) : D_2 \to \wp(D_1)$
$\quad | \ (X : D_2 \to \wp(D_1) \in \rho_4(\phi)) \wedge (X' : D_2 \to \wp(D_1) \in \rho_4(\phi'))\}$

In this case, the set of expressions produced contains at most one element per pair of elements in the cross product of the refinements of ϕ and ϕ' where both elements have the same type.

Using rules of this form, a set of alternative models is generated by taking the cross product of the sets of refinements of each object level constraint. Reconsider the model represented by constraints (6), (8) and (9). The presence of two representations of the BIBD relation in this model underlines the importance of point 3 above.

Point 3 is achieved by adding channelling constraints to maintain consistency between these two representations. Recall that the object and target-level representations are linked using axioms (5) and (7). In this case, the axioms produce:

$$\forall \langle b, v \rangle : B \times V \cdot BIBD(b, v) \leftrightarrow v \in BIBD_{MB}[b] \tag{10}$$
$$\forall \langle b, v \rangle : B \times V \cdot BIBD(b, v) \leftrightarrow b \in BIBD_{MV}[v] \tag{11}$$

Using constraints (10) and (11), it is simple to construct channelling constraints to connect the two target-level objects:

$$\forall \langle b, v \rangle : B \times V \cdot v \in BIBD_{MB}[b] \leftrightarrow b \in BIBD_{MV}[v]$$

In order to illustrate this set-based refinement mechanism, reconsider the refinement of constraint (1). We proceed in an inside-out manner to show how the refined objects are formed and composed. First, RELATION4 is used to refine the BIBD relation itself into a set of three target-level objects:

$$\begin{aligned} \rho_4(BIBD) = \{ & \lambda d_1, d_2.BIBD_{M01}[d_1, d_2] : B \times V \to 01, \\ & \lambda d.BIBD_{MB}[d] : B \to \wp(V), \\ & \lambda d.BIBD_{MV}[d] : V \to \wp(B)\} \end{aligned}$$

Note how the conditions of PROJECTION4A capture successfully the situation described in Section 3.1: only two of these three objects are permitted to be projected onto $b : B$. Hence, the unsupported refinement of projecting $BIBD_{MV} : V \rightarrow \wp(B)$ onto an element of B is not made.

$$\rho_4(BIBD(b, \_)) = \{\lambda v.BIBD_{M01}[b, v] : V \rightarrow 01, \\ BIBD_{MB}[b] : \wp(V)\}$$

The set-based version of the cardinality rule is introduced before continuing:

CARDINALITY4: $\rho_4(|\phi|) = \{\sum_{d \in D} X(d) : \text{nat} \mid X : D \rightarrow 01 \in \rho_4(\phi)\} \ \cup \\ \{|X| : \text{nat} \mid X : \wp(D) \in \rho_4(\phi)\}$

A larger sub-expression of constraint (1) can now be refined:

$$\rho_4(|BIBD(b, \_)|) = \{ \textstyle\sum_{v \in V} BIBD_{M01}[b, v] : \text{nat}, \\ |BIBD_{MB}[b]| : \text{nat}\}$$

The EQUALITY4, CONST/PARAM4, and QUANTIFIER4 rules follow a similar pattern, leading to the full refinement of constraint (1):

$$\rho_4(\forall b : B \cdot |BIBD(b, \_)| = k) = \{ \forall b : B \cdot \textstyle\sum_{v \in V} BIBD_{M01}[b, v] = k : 01, \\ \forall b : B \cdot |BIBD_{MB}[b]| = k : 01\}$$

This example has shown how alternative refinements of abstract expressions can be composed via dynamic type checking.

4 Identifying and Breaking Symmetry

Experienced modellers know that the effective solution of many problems requires identifying and breaking many of the symmetries that are present in a model of the problem. We claim that symmetries in models arise from (at least) two sources: some symmetries may be present in the original statement of the problem and other symmetries may be introduced by the refinement process. Though it is important to deal with both types of symmetry, dealing with the latter type is central to the study of refinement.

Refinement introduces symmetries when it creates a model that distinguishes between objects that are not distinguished in the original problem formulation. For example, the original statement of the BIBD problem refers to a set B of blocks and a set V of objects, but does not distinguish among the elements of each set; no particular block or object is named. In the 01 matrix model given in Section 3.1, the blocks and objects serve as indices of the two dimensional matrix $BIBD_{M01}$. Indices of a matrix are distinguished; one index refers to the first row/column of a matrix, another refers to the second row/column, and so forth. Hence $BIBD_{M01}$ has row and column symmetry: its rows can be interchanged and its columns can be interchanged [2].[2]

[2] More precisely, an assignment of values to the variables of the matrix is a solution to the problem if and only if it is a solution after swapping the values assigned to any two rows/columns.

We now show how our refinement rules can be enhanced to to recognise the symmetries that they introduce and to incorporate appropriate symmetry-breaking techniques into the models they produce. In the case of the BIBD example, it is relatively simple to infer that B and V each contain indistinguishable elements and hence a refined matrix indexed by B or V (or both) will have index symmetry. The declarations of B and V as types do not name their element; hence there is no way that an individual element of either B or V can be distinguished by the problem constraints.

We enhance our refinement operator so that with each alternative refinement that it generates it associates a set of declarations, a set of annotations and a set of constraints that are to be added to the model. For the purpose of symmetry breaking, the refinement operator produces annotations that identify the symmetry that is to be broken. A later stage of refinement can then use these annotations to break the symmetry, either by adding symmetry-breaking constraints to the model or by using a search technique that dynamically prunes symmetric branches of the search space. Hence, our new refinement operator ρ_5, generates a set of (up to) 4-tuples, where each 4-tuple is of the form

– e `letting` *declarations* `annotate` *annotations* `such that` *constraints*

Here e is the refined expression (as in previous rules), *declarations* is the set of constant declarations to add to the model, and *annotations* and *constraints* are the set of annotations and constraints to add to the model. For the present purpose of symmetry detection we add only annotations. However, in the more complex example considered in the next section, all three facilities are necessary.

As an example, consider the Relation5 rule:

Relation5: $\rho_5(R : \wp(D_1 \times D_2)) =$
$\{\ \lambda d_1, d_2.R_{M01}[d_1, d_2] : D_1 \times D_2 \rightarrow 01$
`annotate` $\{\text{symIndex}(R_{M01}, D_1, \text{equivClasses}(D_1)),$
$\text{symIndex}(R_{M01}, D_2, \text{equivClasses}(D_2))\},$
$\lambda d.R_{MD_1}[d] : D_1 \rightarrow \wp(D_2)$
`annotate` $\{\text{symIndex}(R_{MD_1}, D_1, \text{equivClasses}(D_1))\},$
$\lambda d.R_{MD_2}[d] : D_2 \rightarrow \wp(D_1)$
`annotate` $\{\text{symIndex}(R_{MD_2}, D_2, \text{equivClasses}(D_2))\}\}$

As per Relation4 (see Section 3.2), when Relation5 is applied to a binary relation, it produces a set of three alternative refinements: a two-dimensional 0/1 matrix, and two one-dimensional matrices of sets. Since this refinement has the potential to introduce symmetry, the rule checks whether symmetry has been introduced by checking whether one or more equivalence classes can be formed from the elements of the index set(s) of the matrix. If so, the model is annotated to reflect the fact that a matrix has symmetry in one or more of its indices. A decision can then be taken as to how to break this symmetry. If more than one matrix in a model is indexed by the same symmetrical set, symmetry must be broken consistently in all cases [2]. It should be clear that the symmetry declaration is local to an element of the set of refinements of R: R_{MD_2} cannot introduce symmetry on the indices of D_1, whereas R_{M01} can.

We shall not elaborate further on the mechanisms for dealing with symmetry, but instead close by noting that refining a specification that includes indistinguishable objects inevitably introduces symmetry. In its basic conception, an instance of the finite domain constraint satisfaction problem consists of a set of *named* variables, each of which is associated with a finite domain of *named* values. Hence, refining an abstract specification with indistinguishable objects must name, and thus distinguish, the objects, thereby introducing symmetry.

5 Refining Compositional Types

We now describe the refinement of a more complex problem, Social Golfers [6] (Problem 10 at www.csplib.org), which further illustrates compositional refinement and symmetry detection. An English definition of the problem is as follows:

> Schedule p golfers into groups of size s for each of w weeks, in such a way that any two golfers play in the same group at most once.

The schedule can be represented by a set S, of fixed size w, of regular partitions, so this problem can be expressed as follows:

> `given` p, w, s:nat
> `letting` G `be` type-of-size p
> `find` $S : \wp_w(\text{regpart}\ G\ s)$
> `such that` $\forall \{R, R'\} \subseteq S \cdot \forall g \in R, g' \in R' \cdot |g \cap g'| \leq 1$

where $R : \wp(\text{regpart}\ G\ s)$, and $R' : \wp(\text{regpart}\ G\ s)$ are regular partitions in S; $\{R, R'\}$ is a fixed-size subset of S; and $g : \wp_s(G)$ and $g' : \wp_s(G)$ are groups in a regular partition. Hence, any two groups have at most one player in common.

This problem is specified in terms of a sized relation, the refinement of which has not been discussed previously. Initially, therefore, consider the following refinement rule in the style of ρ_5, which assumes that the arguments of the sized relation have atomic types. For every sized relation $R : \wp_n(D)$, R_{MI} denotes a 1-dimensional matrix whose elements are of type D and that is indexed by I, a locally introduced type of size n:

SizedReln5: $\rho_5(R : \wp_n(D)) =$
$\{\ \lambda i.R_{MI}[i] : I \rightarrow D$
`letting` I `be` type-of-size n
`annotate` symIndex$(R_{MI}, I)\}$

Note that, via I, we have distinguished the index of each element of R, whereas the original relation representation did not. This introduces a symmetry which is recorded in the annotations of this refinement. It is also possible to refine R to a 01 matrix, with constraints to ensure that a total of n tuples are admitted. For brevity, these rules are omitted, since the above refinement is the most straightforward when the arguments of the relation have molecular types, as per $S : \wp_w(\text{regpart}\ G\ s)$ from the Social Golfers example.

General compositional refinement, which must support arbitrarily nested molecular type expressions, requires a recursive application of the refinement rules to ensure that the input specification is fully refined. This approach is embodied by our new refinement operator, ρ_6. The general version of the sized relation refinement rule provides an example. We saw how SizedReln5 refines a relation $R : \wp_n(D)$ to a one-dimensional matrix $R_{MI} : I \to D$, where I is a locally introduced type of size n. The generalised rule refines a relation $R : \wp_n(\tau)$, where τ is an arbitrary type expression, to a one-dimensional matrix $R_{MI} : I \to \rho_6(\tau)$. Instead of refining each element of R_{MI} individually, which could lead to an exponential number of refinements of R, we refine an arbitrary element of τ and ensure that every element of R_{MI} is refined identically:

SizedReln6: $\rho_6(R : \wp_n(\tau)) =$
$\{\ \lambda i.R_{MI}[i] : I \to \tau'$
$\quad$ `letting` I `be` type-of-size $n, \forall i : I \cdot dcls[R_{MI}[i]/e']$
$\quad$ `annotate` symIndex$(R_{MI}, I), \forall i : I \cdot anns_1[R_{MI}[i]/e']$
$\quad$ `such that` $\forall i : I \cdot cstr[R_{MI}[i]/e']$
$\quad|\quad (e' : \tau'$ `letting` $dcls$ `annotate` $anns_1$ `such that` $cstr) \in \rho_6(e : \tau)\}$

More concretely, to refine τ we introduce a variable e to stand for an arbitrary element of τ and refine it to give $e' : \tau'$. The declarations, annotations and constraints associated with e' are lifted to hold for all elements of τ', with the appropriate element of R_{MI} substituted for e' (where $exp[x/y]$ denotes the substitution of x for all occurrences of y in exp). Refining an element of an atomic type yields an unchanged type and no declarations, annotations or constraints.

The regular partition type is also central to the specification of the Golfers problem. For every regular partition $R : regpart\ \tau\ n$, let $R_f : \tau \to P$ denote a function from τ to a locally declared type P (for partition identifiers). Since R_f is not atomic, it too must be refined. The refinement of R_f is denoted R'_f.

RegPartition6: $\rho_6(R : regpart\ \tau\ n) =$
$\{\ R'_f$
$\quad$ `letting` P `be` type-of-size $|\tau|/n, dcls_1, dcls_2$
$\quad$ `annotate` $anns_1, anns_2$
$\quad$ `such that` $C \wedge cstr_1 \wedge cstr_2$
$\quad|\quad R'_f$ `letting` $dcls_1$ `annotate` $anns_1$ `such that` $cstr_1$
$\qquad \in \rho_6(R_f : \tau \to P) \wedge$
$\quad C$ `letting` $dcls_2$ `annotate` $anns_2$ `such that` $cstr_2$
$\qquad \in \rho_6(\forall x : P \cdot |R'^{-1}_f(x)| = n)\}$

That is, if R is a regular partition of τ into disjoint subsets of size n, its refinement is R'_f, the refinement of a total function R_f, mapping τ into the new type P. To ensure that R'_f accurately represents R, we introduce and refine the constraint $\forall x : P \cdot |R'^{-1}_f(x)| = n$ (i.e. all partitions have size n, following refinement).

Now let us consider an example illustrating these two rules. If RegPartition6 is applied to an element $R \in S$ in the Social Golfers problem, and we assume that the function representing the partition is further refined to a one-dimensional

matrix, R_{MP} of sets of golfers indexed by P, the result is as follows. For reasons of space, refinement rules for functions are omitted. Note, however, that the symmetry introduced by indexing the matrix R_{MP} by P, a locally introduced type with indistinguishable elements, is detected and recorded.

$$\begin{aligned}\rho_6(R : regpart\, G\, s) = \{\ & R_{MP} : P \to \wp(G) \\ & \texttt{letting}\ P\ \texttt{be}\ \text{type-of-size}\ |G|/s \\ & \texttt{annotate}\ \text{symIndex}(R_{MP}, P) \\ & \texttt{such that}\ \forall x : P \cdot |R_{MP}[x]| = s \wedge \\ & \forall \{x, x'\} : \wp(P) \cdot R_{MP}[x] \cap R_{MP}[x'] = \emptyset\}\end{aligned} \tag{12}$$

Now, if SizedReln6, and recursively RegPartition6, are applied to the example, S becomes a matrix S_{MW}, indexed by weeks in W, of matrices representing the weekly schedules.

$$\begin{aligned}\rho_6(\ & S : \wp_w(\text{regpart}\ G\ s)) = \{ \\ & \lambda i.S_{MW}[i] : W \to (P \to \wp(G)) \\ & \texttt{letting}\ W\ \texttt{be}\ \text{type-of-size}\ w, P\ \texttt{be}\ \text{type-of-size}\ |G|/s \\ & \texttt{annotate}\ \text{symIndex}(S_{MW}, W), \forall i : W \cdot \text{symIndex}(S_{MW}[i], P) \\ & \texttt{such that}\ \forall i : W \cdot \forall p : P \cdot |S_{MW}[i][p]| = s \wedge \\ & \forall i : W \cdot \forall \{x, x'\} : \wp(P) \cdot S[i][x] \cap S[i][x'] = \emptyset\}\end{aligned} \tag{13}$$

We now introduce a new universal quantifier rule for the case where the bound variable is a member of an ordinary set expression.

ForallElem6: $\rho_6(\forall x \in Y \cdot C) =$
$\{\ \forall x' \in Y' \cdot C'$
$\quad \texttt{letting}\ dcls_1, dcls_2$
$\quad \texttt{annotate}\ anns_1, anns_2$
$\quad \texttt{such that}\ cstr_1 \wedge cstr_2$
$\mid\ x' \in Y'\ \texttt{letting}\ dcls_1\ \texttt{annotate}\ anns_1\ \texttt{such that}\ cstr_1 \in \rho_6(x \in Y) \wedge$
$\quad C'\ \texttt{letting}\ dcls_2\ \texttt{annotate}\ anns_2\ \texttt{such that}\ cstr_2 \in \rho_6(C[x'/x])\}$

That is, we refine a finitely universally quantified variable x by refining the binding expression $x \in Y$, then substituting x', a refinement of x, for x in the constraint C, then refining C. The substitution of x' for x prior to the refinement of C ensures that each occurrence of x in C is refined in the same way. The definitions of the Element6 rules are omitted for brevity.

Applying ForallElem6 to our example results in the following constraint:

$$\begin{aligned}\rho_6(\ & \forall g \in R, g' \in R' \cdot |g \cap g'| \leq 1) = \{ \\ & \forall g \in \{R_{MP}[p] \mid p : P\}, g' \in \{R'_{MP}[p] \mid p : P\} \cdot |g \cap g'| \leq 1 \\ & \texttt{letting}\ P\ \texttt{be}\ \text{type-of-size}\ |G|/s \\ & \texttt{annotate}\ \text{symIndex}(R_{MP}, P) \\ & \texttt{such that}\ \forall p : P \cdot |R_{MP}[p]| = s \wedge \forall \{x, x'\} : \wp(P) \cdot R_{MP}[x] \cap R_{MP}[x'] = \emptyset \\ & \wedge \forall p : P \cdot |R'_{MP}[p]| = s \wedge \forall \{x, x'\} : \wp(P) \cdot R'_{MP}[x] \cap R'_{MP}[x'] = \emptyset\}\end{aligned} \tag{14}$$

A similar rule, ForallSubset6, refines universally quantified expressions where the binding expression is a subset. Hence, assuming rules for refining cardinality and intersection expressions similar to those presented in the previous sections, the Social Golfers specification can be refined via the application of ForallSubset6:

$$
\begin{array}{l}
\texttt{given } p, w, s\text{:nat} \\
\texttt{letting} \\
\quad G \texttt{ be} \text{ type-of-size } p,\ W \texttt{ be} \text{ type-of-size } w,\ P \texttt{ be} \text{ type-of-size } |G|/s, \\
\texttt{annotate}\, \text{symIndex}(S_{MW}, W),\ \forall i : W \cdot \text{symIndex}(S_{MW}[i], P) \\
\texttt{find}\ S_{MW} : W \rightarrow (P \rightarrow \wp(G)) \\
\texttt{such that} \\
\quad \forall \{R_{MP}, R'_{MP}\} \subseteq \{S_{MW}[i] \mid i : W\} \cdot \\
\qquad \forall g \in \{R_{MP}[p] \mid p : P\}, g' \in \{R'_{MP}[p] \mid p : P\} \cdot |g \cap g'| \leq 1 \wedge \\
\quad \forall i : W \cdot \forall p : P \cdot |S[i][p]| = s \wedge \\
\quad \forall i : W \cdot \forall \{x, x'\} : \wp(P) \cdot S[i][x] \cap S[i][x'] = \emptyset
\end{array}
$$

Having refined the binding expression of the outer universal quantifier, which causes R and R' to be refined to the matrices R_{MP} and R'_{MP}, the refinement of the remainder is much simpler. This is because R_{MP} is substituted for R (and similarly for R'_{MP}) beforehand. Several other refinements are possible from the input specification, according to how S and the regular partition are refined. These are not given for reasons of space. The model developed here is close to that developed by hand by Stefano Novello[3].

6 Discussion of our Specification Language

Now that we have seen how our compositional refinement system operates, we can consider how the demands of effective refinement have driven the design of the specification language. 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 [8] and ESRA [3], each of which was designed to be more abstract than its predecessors.

One demand is that the specification language must be powerful enough to enable important distinctions to be drawn. Consider, for example the specifications of the BIBD problem and the Social Golfers problem. The decision variables in these two problems are, respectively, a relation and a set of partitions (that is, a set of sets of sets). Though each of these decision variables can be refined to a matrix of decision variables, the matrices have different symmetries. As our specification language supports both relations and sets of partitions, specifications can distinguish between these two types, thus providing the refinement system with the opportunity to choose the appropriate symmetry breaking. Many languages, such as OPL, do not support sets or relations, so both problem specifications must employ matrices. Though ESRA is a more abstract

[3] Available at http://www.icparc.ic.ac.uk/eclipse/examples/

language, which supports sets and relations, it does not support sets of sets, so a natural specification of both problems would have decision variables of type relation. Therefore, in any of these languages the identification of symmetry is the responsibility of the user, not the compiler or refinement system.

Effective refinement also demands that the specification language must provide sufficient abstraction so that a specification need not make unnecessary distinctions, such as distinguishing between objects that should be indistinguishable. The consequence of using such a language is that fewer symmetries are contained in the initial specification, but more are introduced by refinement, which we believe can be identified and broken automatically. This point is clearly exemplified in our treatment of the BIBD problem. The initial specification does not name the elements of B or V; names are introduced by the refinement process, which simultaneously recognises that this introduces symmetry into the model. We know of no constraint specification language that supports sets of unnamed objects. Without this facility, the job of recognising the interchangeability of the arbitrarily-named objects is the responsibility of the user.

7 Conclusion

This paper has provided substantial support for the conjecture that constraint satisfaction models can be constructed by compositional refinement of an abstract specification. This is an important claim; if models cannot be constructed compositionally, then it is hard to see how one could construct an automated refinement system that handled a broad class of specifications. This is also a significant result; because the decisions involved in constructing a model are not independent, it is difficult to identify the manner in which models can be composed and to capture this in a set of rules.

Before safely concluding that refinement is compositional it would be necessary to show that a wide range of models appearing in the literature can be constructed by compositional rules. Doing this would require developing a more expressive specification language, identifying a broader range of structures that are employed by expert modellers, and formulating a more extensive set of reformulation rules.

A set of broad-coverage refinement rules would lie at the heart of an automated system for the competence stage of modelling. But such a system would need much more. As discussed in Section 4, the system would need to identify symmetries in the models it generates. Once a model is produced, a system should be able to improve it through certain transformations, just as human expert modellers do. Such transformations include adding constraints that are implied by those that are already in the model but which could potentially increase pruning of the search space; removing redundant constraints (those that result in no extra pruning of the search space); replacing common subexpressions with an auxiliary variable to increase pruning [5]; and replacing constraints with their logical equivalents. We have investigated transformations such as these, resulting in the development of an automated system called CGRASS [4]. How-

ever, we have only considered applying the transformations to problem instances. The difficult challenge is formulate corresponding transformations that can be applied to problem classes.

Acknowledgements This work is partially supported by UK-EPSRC grant no. GR/N16129. We thank Brahim Hnich and Toby Walsh for useful discussions.

References

1. B.M.W. Cheng, K.M.F. Choi, J.H. Lee, and J.C.K. Wu. Increasing constraint propagation by redundant modelling. *Constraints*, 4(2):167–192, 1999.
2. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Proc. 8th International Conference on Principles and Practice of Constraint Programming*, pages 462–476, 2002.
3. P. Flener, J. Pearson, and M. Ågren. The syntax, semantics, and type system of ESRA. Technical Report ASTRA, Uppsala University, 2003.
4. A.M. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In B. O'Sullivan, editor, *Proc. Joint Workshop of the ERCIM/CologNet area on Constraint Solving and Constraint Logic Programming (LNAI 2627)*, pages 15–30, 2002.
5. W. Harvey and P. J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.
6. M. Sellmann and W. Harvey. Heuristic constraint propagation. In *Proc. 4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR)*, pages 191–204, 2002.
7. H.D. Sherali and J.C. Smith. Improving discrete model representations via symmetry considerations. In *Proc. International Symposium on Mathematical Programming*, 2000.
8. P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
9. T. Walsh. Constraint patterns. In F. Rossi, editor, *Proc. 9th International Conference on Principles and Practice of Constraint Programming*. Springer, 2003.