Refining Abstract Specifications of Constraint Satisfaction Problems

Alan M. Frisch Ian Miguel
AI Group, Dept. Computer Science
University of York
York, England
{frisch, ianm}@cs.york.ac.uk

Toby Walsh
Cork Constraint Computation Center
University College Cork
Cork, Ireland
tw@4c.ucc.ie

1 Introduction

Many and diverse search problems have been solved with great success using constraint programming [5]. However, in order to apply constraint programming tools to a particular domain, the problem must be *modelled* as a constraint satisfaction or optimisation problem. Such a problem consists of a set of decision variables, each with an associated domain of potential values, plus an optional objective function. Constraints that specify the allowed combinations of assignments of values to variables are then posted. A solution is an assignment to all the variables satisfying all the constraints (that optionally maximises or minimises the objective function).

Typically, the modelling process begins with an *abstract* natural language description of a problem. This formulation is progressively *refined* into a *concrete* constraint program. Since constraints provide a rich language, typically many alternative concrete models exist for a given problem, some of which will prove to be more effective than others as input to a constraint solver. Although existing constraint toolkits make writing down a model easier once one is in mind, very little help is available for formulating a good model in the first place or for reformulating an existing model to make it more efficient. Hence, modelling is one of the most important yet difficult stages in solving a problem.

This paper is concerned with automating the task of refining an abstract problem formulation. As well as substantially reducing the expertise needed for modelling, automated refinement will enable a systematic exploration of alternative models which can be evaluated and, if the models have complementary strengths, combined.

A further benefit is in dealing with symmetry. At an abstract level, symmetrical objects are typically not distinguished. However, refinement often introduces symmetries by distinguishing indistinguishable objects. For example, a concrete representation of a set is a list. Many lists can be used to represent the same set, all of which can be considered symmetrical. If one such list, where the elements are ordered, is chosen the symmetry is broken. Hence, by recognising when symmetries are introduced, appropriate symmetry breaking constraints can be added during refinement. In the example, a constraint would be added to insist that the list is ordered. This is significantly more efficient than trying to detect and break symmetry in a concrete formulation.

2 Refinement Rules

Standard constraint satisfaction is founded on the use of variables whose domains are composed of atomic objects such as integers or strings. Consider an abstract language where variables' domains consist of relations over finite domains and which supports constraints on such variables. Since constraint solvers typically do not support abstract relations (although the ESRA [2] and F [4] languages are addressing this issue), in order to translate this type of problem into a constraint program, it must be refined into standard CSP variables and constraints. Typically, more than one choice of standard variables and domains is appropriate, with some choices supporting only a subset of the constraints (just as some data structures support some operations, but not others).

The abstract relation language is briefly described as follows. A unary relation is referred to simply as a set. An n-ary relation, R, over n arguments, $A_1, A_2, ..., A_n$ is denoted: $R : A_1 \times A_2 \times ... \times A_n$. The Boolean test whether a certain tuple is in R is denoted by $R(a_1, a_2, ..., a_n)$, where $a_1 \in A_1, a_2 \in A_2$, etc. In the following R and R' are the names of n-ary relations and ϕ and ϕ' are arbitrary expressions that denote n-ary relations. The domain of an n-ary relation is considered to be $X_1 \times X_2 \times \cdots \times X_n$. Required operations include:

- **Union/Intersection**: Relations are treated as sets of tuples. Hence, the union and intersection of two relations are simply the union and intersection of their sets of tuples respectively.
- **Cardinality**: The number of tuples in a relation, denoted $|R|$.
- **IsEmpty**: IsEmpty(R) returns true if R admits no tuples, false otherwise.

Furthermore, some constraints are required, including:

- **Equality**: $R = R'$ means that R and R' contain an identical set of tuples.
- **Subset**: $R \subseteq R'$ means that the set of tuples in R must be a subset of or equal to those in R'.

Strongly typed rules are used to refine a formulation from a higher to a lower level of abstraction. The rules are compositional. This means, for example, that if the projection of a relation and the intersection of two relations can be refined, then the projection of the intersection can also be refined. A set of rules is created for each alternative refinement of a higher level construct. The following describes two possible examples.

2.1 Alternative 1: 0/1 Matrix

One way to represent an n-ary relation R over $X_1 \times X_2 \times \cdots \times X_n$ is by an n-dimensional matrix R_m of decision variables. The matrix is indexed by members of $X_1 \times$

$X_2 \times \cdots \times X_n$ and each decision variable has the domain $\{0, 1\}$. (0 and 1 are considered to be synonymous with false and true, respectively.)

The relationship between R and R_m is captured by the following axiom:

$$\forall \langle i_1, \ldots, i_n \rangle \in X_1 \times X_2 \times \cdots \times X_n : \\ R(i_1, \ldots, i_n) \leftrightarrow R_m[i_1, \ldots, i_n] = 1$$

ρ is a refinement operator which maps abstract expressions on relation variables to concrete expressions with finite domain variables. It is recursively defined by a set of rules. For brevity, only a subset is described below:

1. $\rho(R) = \lambda\langle i_1, \ldots, i_n \rangle . R_m[i_1, \ldots, i_n]$.
2. $\rho(\phi(i_1, \ldots, i_n)) = \rho(\phi)(\langle i_1, \ldots, i_n \rangle)$
3. $\rho(\phi \cap \phi') = \lambda\langle i_1, \ldots, i_n \rangle . \rho(\phi)(\langle i_1, \ldots, i_n \rangle) \wedge \rho(\phi')(\langle i_1, \ldots, i_n \rangle)$
4. $\rho(|\phi|) = \sum_{\langle i_1, \ldots, i_n \rangle \in X_1 \times \cdots \times X_n} \rho(\phi)(\langle i_1, \ldots, i_n \rangle)$
5. $\rho(\phi \subseteq \phi') = \forall \langle i_1, \ldots, i_n \rangle \in X_1 \times \ldots \times X_n : \rho(\phi)(\langle i_1, \ldots, i_n \rangle) \rightarrow \rho(\phi')(\langle i_1, \ldots, i_n \rangle)$.

As an example, consider refining the expression $|A \cap B|$ where A and B are binary relations over $X_1 \times X_2$:

$\rho(|A \cap B|) = \sum_{\langle i_1, i_2 \rangle \in X_1 \times X_2} \rho(A \cap B)(\langle i_1, i_2 \rangle)$ by (4).

$= \sum_{\langle i_1, i_2 \rangle \in X_1 \times X_2} \lambda\langle i_1, i_2 \rangle . (\rho(A)(\langle i_1, i_2 \rangle) \wedge \rho(B)(\langle i_1, i_2 \rangle))(\langle i_1, i_2 \rangle)$ by (3).

$= \sum_{\langle i_1, i_2 \rangle \in X_1 \times X_2} \rho(A)(\langle i_1, i_2 \rangle) \wedge \rho(B)(\langle i_1, i_2 \rangle)$ by beta reduction.

$= \sum_{\langle i_1, i_2 \rangle \in X_1 \times X_2} \lambda\langle i_1, i_2 \rangle . (A_m[i_1, i_2])(\langle i_1, i_2 \rangle) \wedge \lambda\langle i_1, i_2 \rangle . (B_m[i_1, i_2])(\langle i_1, i_2 \rangle)$ by (1).

$= \sum_{\langle i_1, i_2 \rangle \in X_1 \times X_2} A_m[i_1, i_2] \wedge B_m[i_1, i_2]$ by beta reduction.

2.2 Alternative 2: Matrix of Sets

Another way to represent an n-ary relation R over $X_1 \times X_2 \times \cdots \times X_n$ is by a one dimensional matrix R_{ms} of decision variables. The matrix is indexed by the members of X_1 and each decision variable is a set variable whose members are drawn from $X_2 \times X_3 \times \cdots \times X_n$.

The relationship between R and R_{ms} is captured by the following axiom:

$$\forall \langle i_1, \ldots, i_n \rangle \in X_1 \times X_2 \times \cdots \times X_n : \\ R(i_1, \ldots, i_n) \text{ iff } \langle i_2, \ldots, i_n \rangle \in R_{ms}[i_1]$$

Similarly, some of the refinement rules are:

1. $\rho(R) = \lambda x . R_{ms}[x]$.
2. $\rho(\phi(i_1, \ldots, i_n)) = \langle i_2, \ldots, i_n \rangle \in \rho(\phi)(i_1)$
3. $\rho(\phi \cap \phi') = \lambda x . \rho(\phi)(x) \cap \rho(\phi')(x)$
4. $\rho(|\phi|) = \sum_{i_1 \in X_1} |\rho(\phi)(i_1)|$
5. $\rho(\phi \subseteq \phi') = \forall i_1 \in X_1 : \rho(\phi)(i_1) \subseteq \rho(\phi')(i_1)$.

For comparison, again consider refining the expression $|A \cap B|$ where A and B are binary relations over $X_1 \times X_2$:

$\rho(|A \cap B|) = \sum_{i_1 \in X_1} |\rho(A \cap B)(i_1)|$ by (5)

$= \sum_{i_1 \in X_1} |\lambda x . (\rho(A)(x) \cap \rho(B)(x))(i_1)|$ by (3).

$= \sum_{i_1 \in X_1} |\rho(A)(i_1) \cap \rho(B)(i_1)|$ by beta reduction.

$= \sum_{i_1 \in X_1} |\lambda x . A_{ms}[x](i_1) \cap \lambda x . B_{ms}[x](i_1)|$ by (1).

$= \sum_{i_1 \in X_1} |A_{ms}[i_1] \cap B_{ms}[i_1]|$ by beta reduction.

3 Selecting Refinements

The efficiency of a formulation is intimately connected to the solution procedure with which it is solved. A constraint solver typically has a suite of standard constraints with which it is able to make strong inferences and hence reduce search effectively. Non-standard constraints must be built from simple primitive constraints. In general the solver can make only relatively weak inferences with such composite constraints, resulting in less efficient search.

Hence, an effective formulation is typically composed of standard constraints. However, given a pair of constraints, c and c', which together express an abstract formulation at the concrete level, the following situation is common: c requires a choice of formulation (in terms of variables and their domains) which is incompatible with that required by c'. A solution which has repeatedly proved successful [1] is to use *both* choices of variables and domains and add *channelling* constraints to maintain consistency between the two.

Given such a combined approach, the number of possible alternative formulations rises significantly. Hence, a systematic method of refinement capable of generating and combining many candidate formulations should prove valuable.

4 Next Steps

The refinement rules described here and others like them are being incorporated into the CGRASS automated reformulation tool [3], which currently works at the concrete level of abstraction only. Hence, CGRASS will be able to work at multiple levels of abstraction and produce alternative models automatically. It is also expected that some types of reformulation (such as the addition of implied constraints) will be easier at higher levels of abstraction.

Other components of the solution procedure, such as the choice of search heuristic and the strength of inference available for each constraint, also have a marked effect on the utility of the model. Therefore, an important piece of future work is to examine how a search heuristic can be chosen to suit a given model, and how knowledge about the strength of inference available for each constraint can be incorporated into the reformulation process.

Acknowledgements This project and the second author are supported by EPSRC Grant GR/N16129[1]. The third author is supported by Science Foundation Ireland.

References

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

[2] P. Flener and B. Hnich. The syntax and semantics of esra. Astra report, Department of Information Science, Uppsala University, 2001.

[3] A.M. Frisch, I. Miguel, and T. Walsh. Cgrass: A system for transforming constraint satisfaction problems. In *Proc. ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 23–36, 2002.

[4] B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, Department of Information Science, Uppsala University, 2003.

[5] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[1] `http://www.cs.york.ac.uk/aig/projects/implied/index.html`