CGRASS: A System for Transforming 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
Ireland
tw@4c.ucc.ie

Abstract. Experts at modelling constraint satisfaction problems (CSPs) carefully choose model transformations to reduce greatly the amount of effort that is required to solve a problem by systematic search. It is a considerable challenge to automate such transformations and to identify which transformations are useful. Transformations include adding constraints that are implied by other constraints, adding constraints that eliminate symmetrical solutions, removing redundant constraints and replacing constraints with their logical equivalents. This paper describes the CGRASS (Constraint Generation And Symmetry-breaking) system that can improve a problem model by automatically performing transformations of these kinds. We focus here on transforming individual CSP instances. Experiments on the Golomb ruler problem suggest that producing good problem formulations solely by transforming problem instances is, generally, infeasible. We argue that, in certain cases, it is better to transform the problem class than individual instances and, furthermore, it can sometimes be better to transform formulations of a problem that are more abstract than a CSP.

1 Introduction

Constraint satisfaction is a successful technology for tackling a wide variety of search problems including resource allocation, transportation and scheduling. Constructing an effective model of a constraint satisfaction problem (CSP) is, however, a challenging task as new users typically lack specialised expertise. One difficulty is in identifying transformations, which are sometimes complex, that can dramatically reduce the effort needed to solve a problem by systematic search (see, for example, [15]). Such transformations include adding constraints that are implied by other constraints in the problem, adding constraints that eliminate symmetrical solutions to the problem, removing redundant constraints and replacing constraints with their logical equivalents. Unfortunately, outside a highly focused domain like planning (see, for example, [6]), there has been little research on how to perform such transformations automatically.

Our initial focus is on transforming individual CSP instances. The CGRASS system (Constraint GeneRation And Symmetry-breaking) is described and illustrated via the Golomb ruler problem [15], a difficult combinatorial problem with many applications. Our results suggest that making transformations to single problem instances alone is not practical on large instances. This can be remedied in two ways. First, by operating

on a parameterised formulation of a problem *class*, which can be much more compact than the large instances in the class. In addition, every inference made holds for all instances of the class. A second improvement is to reason with formulations at higher levels of abstraction. We can construct *refinement* rules which, when applied to abstract formulations, produce effective concrete formulations. We conjecture that certain inferences will be easier at higher levels of abstraction. CGRASS provides a basic platform to achieve these goals. The rules we have developed form a template for creating new rules for similar reasoning about problems and at higher levels of abstraction.

2 Architecture of CGRASS

The implementation of CGRASS discussed here takes a problem instance as input. A problem instance consists of a finite set of domain variables, each with an associated finite domain (either explicitly or as bounds) and constraints over these variables. Output is created in the same simple format. Hence, very little effort is necessary to translate CGRASS' output into the required input for a variety of existing solvers.

CGRASS captures common patterns in the hand-transformation of constraint satisfaction problems in *transformation rules*. A transformation rule is a condition-action pair in the style of a production rule system (see [11], chapter 5). A rule-based architecture offers several potential advantages for the task of transforming CSPs automatically. Rules can be given very strong pre-conditions to limit the transformations to those that are likely to produce a problem that is simpler to solve. Furthermore, rules can act at a very high level. For example, they can perform complex rewriting, simplifications, and transformations. Such steps might require long fine-grained sequences of transformations to justify at the level of individual inference rules.

The structure of a CGRASS transformation rule can be demonstrated using an example. Consider the following pair of constraints, where the domains of x_1, x_2, x_3 are all $\{1, \ldots, 9\}$ and the domain of y is $\{1, \ldots, 100\}$:

$$x_1 + x_2 + x_3 = y$$
$$\text{allDifferent}(x_1, x_2, x_3).$$

Reasoning about the domains of x_1, x_2, x_3 and the fact that they are all-different gives:

$$6 \leq x_1 + x_2 + x_3$$
$$x_1 + x_2 + x_3 \leq 24$$

Substituting these inequalities back into the original equation clearly provides stronger bounds on y than bounds consistency alone.

The `allDiffSum` rule presented in Figure 1 achieves this transformation. The form of CGRASS' rules is adapted from the *methods* used to capture common proof patterns in proof planning [2]. The condition part of the rule consists of the input and pre-condition fields. The action part comprises the post-conditions and the add and delete lists, as explained below.

The input field specifies a pattern against which a subset of the constraint set must match before the rule can be applied. CGRASS uses a rich pattern matching language

Name: `allDiffSum`
Input: `algebraic(Constraint), allDifferent(Vars)`
Pre-conditions: `containsSimpleSum(Constraint, Sum),`
`SumVars = involvesVars(Sum),`
`supersetEq(Vars, SumVars)`
Post-conditions: `LB = allDiffLB(Sum),`
`UB = allDiffUB(Sum)`
Add: `LB <= Sum, Sum <= UB`
Delete:
Explanation: `"Given ",Sum,"such that all variables`
`involved are all-different, we can infer",`
`LB<=Sum," and ",Sum<=UB`

Fig. 1. The `allDiffSum` method.

specialised to reasoning about sets of constraints. This includes the ability to restrict matching to objects of certain types, such as a single constant or variable. CGRASS also supports several more powerful matching instructions, such as `algebraic(C)` and `allAlgebraic(C)`, which match C with an arbitrary algebraic constraint and the algebraic subset of the current constraint set respectively. In the example, `Constraint` matches $x_1 + x_2 + x_3 = y$ and `Vars` matches $\{x_1, x_2, x_3\}$.

Pre-conditions must be met before the post-conditions can be executed. Conditions are composed from primitive Boolean functions, such as $=$ (which also performs assignment), $!=$ (disequality) and $<$, as well as specialised functions which can be used to perform more complex operations on the input constraints. The specialised functions are written directly in Java (the native language of CGRASS), hence there is no restriction as to the operations that these functions can perform. The current set of specialised functions is readily extensible by writing new Java functions, although this does require some knowledge of how CGRASS works internally.

In the example of Figure 1, `containsSimpleSum(Constraint, Sum)` unifies `Sum` with a sub-term of `Constraint` composed of a sum with integer coefficients, i.e. $x_1 + x_2 + x_3$. `involvesVars(Sum)` returns the list of variables involved in `Sum`, which must be a subset or or equal to `Vars`. Finally, `allDiffLB(Sum)` and `allDiffUB(Sum)` calculate lower and upper bounds of a sum of all-different variables by considering the bounds of each individual variable involved. These bounds are used to add the two new constraints.

The size of the constraint set on which CGRASS operates neither increases nor decreases monotonically. This is because some of CGRASS' rules add new constraints, whereas others replace a constraint by a tighter one, or eliminate redundant constraints. For this reason, we replace the rule 'output' field normally used by proof planning by 'add' and 'delete' lists as used in classical planning. Both fields contain sets of constraints. In the case of the delete list, each element of this set is matched against and removed from the current constraint set.

In order for the user to see how a new model was derived, CGRASS' rules construct textual explanations of their application. For flexibility, sub-sections of the text can be predicated on the state of Boolean variables local to the rule.

2.1 Operation

CGRASS performs pure forward chaining, transforming one set of constraints into another. CGRASS' rules are sorted in descending order of priority. Given an input constraint set, CGRASS iterates over the sorted rule list to find the highest priority applicable rule. This rule is used to transform the constraint set, which is then used as input to the same process. Currently, CGRASS terminates when none of its rules are applicable. This is possible because of the strength of the pre-conditions attached to each rule. As the rule database grows in size and complexity, however, this may be insufficient. At some point the decision must be made to stop making transformations and start searching for a solution. We may in the future have to add an executive in the style of a proof critic [10] which terminates CGRASS when future rewards look poor.

Non-monotonicity carries with it the danger of looping. Unless a rule deletes some of its input constraints, its preconditions continue to hold. Hence, the rule can repeatedly fire *ad infinitum*. To avoid the problem CGRASS employs a history mechanism, which maintains a record of all constraints that have been added to the constraint set. Rules are not allowed to add a constraint which was added to the constraint set previously, even if the constraint in question was subsequently removed. The intuition behind this approach is that a constraint is removed either if it is redundant or it is transformed into some more useful form. Restoring a previously removed constraint is therefore a retrograde step.

2.2 Normalisation

CGRASS transforms the constraint set into a normal form at the start of operation and any time the constraint set is modified. The normal form used is inspired by that used in the HARTMATH computer algebra system[1]. This enables CGRASS to deal easily with associative and commutative operators, allowing it to test for simple syntactic equivalence instead of semantic equivalence. Normalisation also reduces the number and complexity of rules needed. For example, inequalities are always rearranged into the form $x < y$ or $x \leq y$. Hence the input to a rule never has to match $y > x$ or $y \geq x$, halving the number of rules in some cases.

We define a total order over the types of expressions that CGRASS supports. The constraint set is transformed into a minimal state with respect to this order. Constants are at the top of the order, followed by variables, fractions, sums and products. Further down the order are constraint types such as equalities, disequations, inequalities and special constraints such as 'all-different'. Expressions of different type are ordered via their position in the order. Expressions of the same type are ordered recursively; each type has an associated rule of self-comparison. The base case is where two constants or two variables are compared. In the former case, the comparison is by value, with least first and in the latter the comparison is lexicographically by name. Sums and products are represented in a 'flattened' form, hence their arguments are simply sorted using the above comparison to maintain a lexicographic order. Similarly, the lexicographically least side of an equation or disequation is forced to be the left hand side.

[1] `http://www.hartmath.org`

As an example, consider the following pair of constraints:

$$x_8 + x_7 \neq x_6 + x_5$$
$$x_4 * 2 + x_3 = 2 * x_1 + x_2$$

Transforming them to normal form results in:

$$x_2 + 2 * x_1 = x_3 + 2 * x_4$$
$$x_5 + x_6 \neq x_7 + x_8$$

Equality is higher in the type order than disequality, hence the re-ordering of the two constraints. The sums are ordered internally and recursively, then re-ordered as appropriate to the constraint.

Simplification procedures consist of the collection of like terms, cancellation and the removal of common factors. Consider the following example:

$$2 * 6 * x_1 + 4 * x_2 = 6 * x_1 + x_3 * 2 * 2 + 6 * x_1$$

Following lexicographical ordering, we collect the constants and occurrences of x_1:

$$4 * x_2 + 12 * x_1 = 4 * x_3 + 12 * x_1$$

Next we perform cancellation:

$$4 * x_2 = 4 * x_3$$

Finally, we remove the common factor:

$$x_2 = x_3$$

Lexicographic ordering and simplification are interleaved until no further change is possible. They reduce the workload of CGRASS substantially, both in providing a syntactic test for equality and avoiding such simplification routines being written as explicit rules. The latter saving is substantial: a larger rule base means more work in matching against the constraint set at each iteration of the CGRASS inference loop.

3 Transformation Rules

We illustrate the transformation rules currently implemented in CGRASS via a small instance of the Golomb ruler problem (available at `http://www.csplib.org` as `prob006`). A Golomb ruler is a set of n ticks at integer points on a ruler of length m such that all the inter-tick distances are unique. Given n, the problem is to minimise m. The longest known optimal ruler has 21 marks and is of length 333. Such rulers have practical applications in radio astronomy and X-ray crystallography [5]. Smith et al. [14] used the Golomb ruler as the basis of an interesting exercise in modelling CSPs and identified a number of implied constraints by hand.

We begin with a concise model of the problem with n ticks represented by variables $x_1, \ldots, x_n$, each with domain $\{0, \ldots, n^2\}$. Note that n^2 is an empirical upper bound,

sufficient for small instances, that we use throughout. It is easy to show that 2^n is a conservative upper bound in general. Alternatively, some solvers allow domains with no upper bound, which are suitable for this type of minimisation problem.

$$\text{minimise: } max_i(x_i)$$
$$\{(x_i - x_j \neq x_k - x_l) \mid i,j,k,l \in [1,n] \wedge (i \neq j) \wedge (k \neq l) \wedge (i \neq k \vee j \neq l)\}$$

The second element is a set of constraints, each element of which is input to CGRASS. Taken literally, this is a poor model. The constraints are quaternary, and will be delayed by most solvers. There is also a large amount of symmetry present, some of which is discussed below. However, it serves to illustrate how CGRASS can make a substantial improvement to a basic model.

We focus on the 3-tick ruler for the purpose of this example. The basic model produces 30 constraints. CGRASS' initial normalisation of the constraint set immediately reduces this number to 12. This is achieved in two ways. Firstly, constraints with reflection symmetry across a disequation are identical following normalisation, hence only one copy is kept. Secondly, multiple constraints can simplify to the same constraint. For example,

$$x_1 - x_2 \neq x_1 - x_3, \text{ and } x_2 - x_1 \neq x_3 - x_1$$

both simplify to $x_2 \neq x_3$.

Hence, a large saving is made before CGRASS has performed any rule application. Table 1 presents the formulation of the problem at this point. The 'minimise' statement is omitted throughout for brevity.

$x_1 \neq x_2$	$x_1 \neq x_3$	$x_2 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

Table 1. 3-tick Golomb ruler. Initial formulation following normalisation.

3.1 Symmetry Breaking

Symmetry is inherent in many CSPs. Given a set of symmetrical variables, it is possible to permute their assignments in a (non)solution to obtain another (non)solution. This can lead to expensive exploration of fruitless branches of the search tree. Hence, it is important to be able to remove or reduce symmetries automatically. CGRASS does this by adding symmetry breaking constraints. Although symmetry breaking constraints are not implied themselves (they do not follow from the initial model), they can be useful for generating further implied constraints. Indeed, often the most useful constraints can be derived only after some or all symmetry has been broken. Hence, CGRASS attempts to detect and break symmetry as a pre-processing step.

CGRASS begins by looking for symmetrical variables, i.e. pairs of variables with identical domains such that, if all occurrences of this pair in the constraint set are exchanged and the constraint set is re-normalised, it returns to its original state. Candidate variables with the same number of occurrences in the constraint set are first grouped together before comparisons are made. The transitivity of symmetry is exploited to minimise the number of pairs of variables that are compared. Efficiency is further improved by making pairwise comparisons of normalised constraint sets.

This process partitions the variables into symmetry classes. The elements of each class are formed into a list, and ordered lexicographically. For each list of variables, say $x_1, \ldots, x_n$, symmetry is broken by adding the constraints

$$x_1 \leq x_2, \quad x_2 \leq x_3, \quad \ldots, \quad x_{n-1} \leq x_n$$

Implied inequalities, such at $x_1 \leq x_3$, are not useful since enforcing bounds consistency on the original set of constraints also enforces generalised arc consistency on the implied constraint.

Symmetry testing on the 3-tick ruler reveals that the variables x_1, x_2 and x_3 are symmetrical. This symmetry can be broken by adding two constraints: $x_1 \leq x_2$ and $x_2 \leq x_3$.

It is also possible to identify symmetries among non-atomic terms. This is potentially an expensive process, hence CGRASS adopts a heuristic approach, only comparing terms that are likely to be symmetrical. These heuristics are based on *structural equivalence*. Two terms are structurally equivalent if they are identical when explicit variable names in each are replaced with a common indistinguishable marker. For example,

$$\frac{x_1}{x_2 * x_3}, \frac{x_4}{x_5 * x_6}$$

become:

$$\frac{\#}{\# * \#}, \frac{\#}{\# * \#}$$

and are therefore structurally equivalent. Each pair of variables, x_1 and x_4, x_2 and x_5, and x_3 and x_6 are exchanged throughout the constraint set before re-normalisation and a check for equivalence. This process does not reveal any further symmetries in the example problem, but is useful in general (see [8], for example).

Working from the symmetry breaking constraints above, CGRASS fires the rule `strengthenInequality`, as presented in Figure 2. This is one example of a number of simple but useful rules to which CGRASS ascribes a high priority during rule selection. Other examples are various instances of the rules `nodeConsistency` and `boundsConsistency` which deal with the filtering of domain elements. These rules are not only cheap to fire, but often result in a reduction in the size of the constraint set. This promotes efficiency by leaving fewer constraints for the more complicated rules to attempt to match against.

Indeed, the `boundsConsistency` rule can now fire, pruning the domains of x_1, x_2 and x_3 according to their strict ordering. This leaves the problem in the formulation as presented in Table 2. Clearly, $x_1 \neq x_3$ is redundant. One could foresee the addition of a relatively simple rule that takes as input a set of strict inequalities and a disequation in order to detect and remove such a redundancy.

Conditions:

1. There exist two expressions, $x \neq y$ and $x \leq y$

Actions:

1. Add a new constraint of the form, $x < y$
2. Delete $x \neq y$ and $x \leq y$

Fig. 2. The `strengthenInequality` rule.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

Table 2. Formulation following symmetry-breaking and bounds consistency.

3.2 Introduce

The model as it stands still contains 9 quaternary constraints. One powerful means of reducing the arity of these constraints is to introduce one or more new variables which the `eliminate` rule (see below) then uses to replace sub-terms within them. Therefore, we have developed the `introduce` rule, as presented in Figure 3. Since this rule

Conditions:

1. There exists a sub-term, *Exp*, containing two or more variables that occurs more than once.
2. *someVariable* = *Exp* is not already present in the constraint set.

Actions:

1. Generate a new variable, x, with domain derived from *Exp*.
2. Add a constraint of the form $x = Exp$.

Fig. 3. The `introduce` rule.

introduces new terms, it has a potentially explosive effect. CGRASS therefore assigns it a very low priority, only attempting to introduce new variables when all the simpler rules, which tend to have a reductive effect, are inapplicable. In addition, complex preconditions are attached to `introduce` to prevent its application unless there is strong

evidence that the new variable will be useful. First of all, we insist that the sub-term, *Exp*, under consideration contains at least two variables; efficiency is unlikely to be gained if Exp contains fewer than two variables.

Secondly, variables which occur frequently have a wider reaching effect when propagation is performed on them. We require *Exp* to occur at least twice in the constraint set before it can be considered for replacement by a variable. Finally, we check that some other variable is not already defined to be equal to *Exp*. If these conditions are met, CGRASS generates a new variable, x, calculating the bounds of its domain from the upper and lower bounds on *Exp*.

The sub-term $x_1 - x_2$ in the example meets the input preconditions of `introduce`. CGRASS introduces a new variable, z_0, with domain {-8 .. 6} and imposes the constraint $z_0 = x_1 - x_2$. In order to make use of z_0, however, the companion `eliminate` rule is necessary.

3.3 Eliminate

We have developed multiple versions of `eliminate`, using equalities (Figure 4) and inequalities to perform Gaussian-like elimination of a particular sub-term.

Conditions:

1. There exists a constraint *Lhs* = *CommonExp* such that *CommonExp* is also present in a constraint, *c*, in the constraint set.
2. Constraint c_{new} is obtained by replacing all occurrences of *CommonExp* by *Lhs* in *c*.
3. The (post-normalisation) size of c_{new} is less than that of *c*.
4. c_{new} is not obviously redundant.

Actions:

1. Add c_{new} to the constraint set.
2. Remove *c* from the constraint set.

Fig. 4. The `Eliminate(equality)` rule.

To ensure that `eliminate` has a reductive effect, the resulting constraint must have a smaller number of constituent terms than the original. Also, we perform simple checks for redundancy such as, in the case of equality, the left hand side being syntactically identical to the right hand side. Finally, when eliminating with equality the original constraint is removed following elimination in order to avoid cluttering the constraint set. `Eliminate` has a higher priority than `introduce`: there is no point in introducing variables for common terms that will be eliminated anyway.

Following the introduction of $z_0 = x_1 - x_2$ in the example, various instances of `eliminate` can fire. For instance, `eliminate(equality)` can be used to substitute z_0 into a number of the quaternary disequations, reducing the complexity of

each. Furthermore, `eliminate(inequality)` eliminates x_1 in favour of x_2 in $z_0 = x_1 - x_2$, using $x_1 < x_2$ to give: $z_0 < 0$. This unary constraint immediately triggers the `nodeConsistency` rule, reducing the domain of z_0 to {-8 .. 0}. The problem is left in the formulation presented in Table 3.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_0 \neq x_2 - x_3$	$z_0 \neq x_3 - x_1$
$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$	$x_1 - x_3 \neq -z_0$
$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$	$x_3 - x_2 \neq -z_0$

Table 3. Formulation following introduction of and elimination with z_0.

CGRASS now introduces, and eliminates with, a further two variables, $z_1 = x_2 - x_3$ and $z_2 = x_3 - x1$. This leads to the much-improved formulation presented in Table 4.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_1 = x_2 - x_3$	$z_2 = x_3 - x_1$
$z_0 \neq z_1$	$z_0 \neq z_2$	$z_1 \neq z_2$

Table 4. Formulation following introduction of and elimination with z_1, z_2.

3.4 All-different

One further rule available to CGRASS is `genAllDiff` which, as its name suggests, attempts to generate an all-different constraint from a clique of not-equals constraints. An all-different constraint is desirable because of the powerful propagation rules available for it within constraint solvers [13]. Since maximal-clique identification is an NP-complete problem, CGRASS uses a fast approximation algorithm [1] to find a clique quickly. Typically this is the maximal clique.

In the example, `genAllDiff` successfully replaces the disequations involving the z variables with a single all-different constraint. This leads to the final problem formulation, as shown in Table 5. This rule has a lower priority than `introduce`: waiting for `introduce` to be exhausted maximises the chance of finding the largest clique of disequations.

4 Results

We compared the performance of Ilog Solver to find and prove optimality on basic and transformed models of 6 instances of the Golomb ruler problem. The transformed models are similar to that presented in Table 5. Results are given in Table 6 and Figure 5.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_1 = x_2 - x_3$	$z_2 = x_3 - x_1$
all-different(z_0, z_1, z_2)		

Table 5. Final formulation.

The smallest instances are so easy to solve that it is not worth the effort of transformation. For larger instances, however, the transformed model becomes significantly easier to solve, with the gap in performance increasing rapidly with n, the number of ticks.

The number of input constraints generated from the basic model also increases significantly with n. Unsurprisingly, this is accompanied by a marked increase in the time required by CGRASS to make the transformations. On the smaller instances tested, this means that the total time for transformation and solution exceeds the time for solution of the basic model alone. However, as n increases, the time required by CGRASS grows more slowly than the time taken to solve the basic model. Hence, at larger (and therefore more interesting) values of n, a net benefit is evident. Furthermore, the effects of a relatively simple implementation are also apparent: as the size of the input grows, CGRASS' rule application rate quickly deteriorates. A more sophisticated implementation that avoids repeated blind traversal of the constraint set would provide a considerable improvement in CGRASS' performance.

One way of overcoming the problem of overwhelming CGRASS on an input instance with many constraints is to use it interactively. Given the final model of the 3-tick ruler, it is not difficult for a human to see how this model could be generalised to a formulation for the entire problem class. The comparative results of the basic and final models presented in Table 6 indicate that the effort expended on such a process could easily be justified as n grows larger. A machine learning tool such as HR [3] or Progol [12] might be used to aid in the generalisation process. Some promising results in this regard are reported in [4].

	Ticks	3	4	5	6	7	8
Basic Model	Size (constraints)	31	133	381	871	1,723	3,081
	Choice-points	12	107	3,637	111,101	4,602,921	160,644,147
	Solution Time	0.01s	0.01s	0.4s	12.3s	1,220s	126,000s
GRASS	Rule Applications	34	120	342	796	1,603	2,917
v1.0	Time	0.1s	1.3s	20s	236s	2,030s	11,600s
Transformed	Choice-points	5	10	76	561	5,402	46,866
Model	Time	0.01s	0.01s	0.01s	0.04s	0.2s	2.6s

Table 6. Results: Golomb ruler. Hardware: 2GHz Athlon XP, 256Mb RAM. Software: Java 1.4.1, Ilog Solver 5.2.

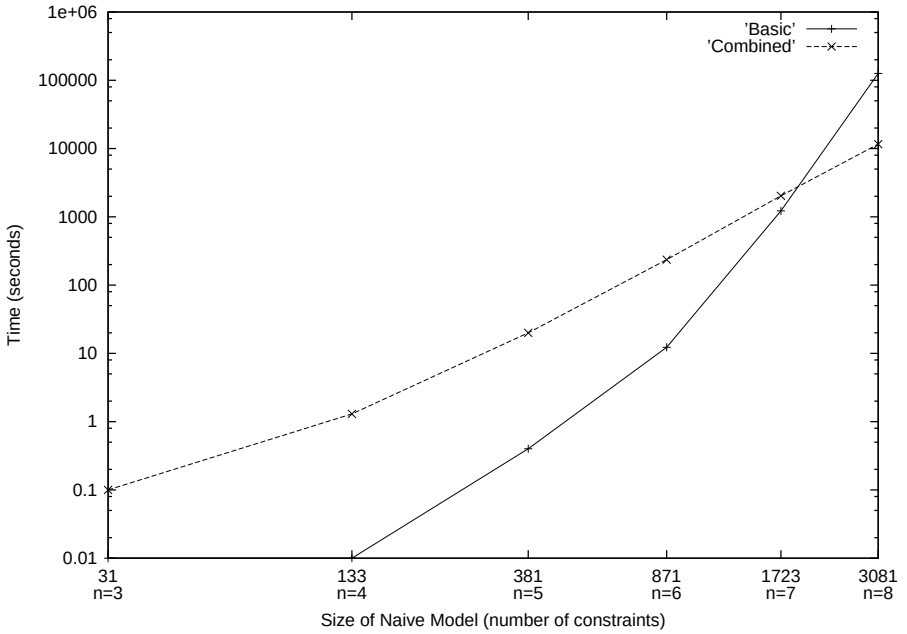

Fig. 5. Shown on a log-log scale are the run-times for solving the basic model versus combined time of CGRASS transformation and solution of transformed model. Each point on the x axis is the problem instance for a given n.

5 Transforming Problem Formulations

The previous section shows that the cost of transforming a problem instance can grow rapidly with its size. A way around this difficulty would be to transform a formulation of the problem rather than a formulation of an instance. Problem formulations usually involve parameters and quantification (or set comprehension) for expressing sets of constraints, so to automate the transformation of problem formulations we must "lift" the CGRASS transformation rules to handle these constructs.

Direct support for quantified constraint expressions would immediately reduce the size of the input in our Golomb ruler example. One such formulation of the problem is:

$$\text{minimise: } max_i(x_i)$$
$$\{(x_i - x_j \neq x_k - x_l) \mid i,j,k,l \in [1,n] \wedge (i \neq j) \wedge (k \neq l) \wedge (i \neq k \vee j \neq l)\}$$

A further benefit is the ability to reason about an entire class of problems rather than particular instances. This is clearly more efficient than repeating a large amount of work for each instance under consideration. However, we should not abandon reasoning about instances altogether; it is likely that some transformations will be valid only for a particular instances of a class. Therefore, we intend to extend—rather than replace—CGRASS' library of transformations to support transformations at the problem level.

Following the introduction of quantified constraint expressions, some transformations remain difficult. For example, one of the key transformations made in Section 4 recognised the symmetry of the tick variables and broke this symmetry via the introduction of weak inequalities. By inspection, detecting this symmetry in the quantified formulation given above is a difficult task. Given the importance of symmetry-breaking, we believe that it is advantageous to start with the problem represented in a significantly more abstract language, much more abstract than, for example, OPL [16].

5.1 Reformulating a Highly Abstract Problem Formulation

This section describes how we might reformulate an abstract description of the Golomb ruler problem, one which is not formally a CSP, into an efficient CSP formulation for input to a constraint solver. Such a reformulation involves two kinds of operations: *refinements* that replace an abstract construct with one that is more concrete, and *transformations*, such as those currently performed by CGRASS, that improve the efficiency of a formulation but do not change its level of abstraction. As we have not yet specified rules to perform these reformulations, our aim here is to illustrate the kind of reformulations that we plan to embody in our rules.

We begin with a natural language description of this problem:

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

We cannot expect CGRASS to work with this level of input, hence the user must make the initial transformation shown below. We chose this formulation because it closely mimics the natural language statement of the problem.

1. Given n find $T \subseteq \{0, \ldots, n^2\}$ subject to:
2. Minimise $max(T)$
3. $|T| = n$
4. $\{\text{distance}(\{x, y\}) \neq \text{distance}(\{x', y'\}) | \{x, y\} \subseteq T, \{x', y'\} \subseteq T, \{x, y\} \neq \{x', y'\}\}$
5. $\{\text{distance}(\{x, y\}) = |x - y| \mid \{x, y\} \subseteq T\}$

Note that $\{x, y\}$ denotes a set of size 2 and that (5) is not part of the natural language specification, but rather encodes user knowledge of the definition of distance.

A crucial feature of this abstract formulation is that it does not distinguish individual ticks in the way that the x_i tick variables of Section 4 do. The x_i variables distinguish ticks by saying "this is the first tick" and "this is the second tick" and so forth. Distinguishing otherwise indistinguishable objects introduces symmetry into a problem which, for the sake of efficiency, must then be broken. Our approach is to start with a formulation that is sufficiently abstract that it does not distinguish otherwise indistinguishable objects. As the formulation is refined to a CSP these objects will necessarily become distinguished since in a CSP all objects are distinguished. Our view is that refinement rules should introduce symmetry breaking constraints at the point they introduce symmetry, that is, at the point they introduce distinction among otherwise indistinguishable objects.

We begin by replacing the occurrences of $distance$ in (4) by the definition given in (5).

6. $\{|x-y| \neq |x'-y'| \mid \{x,y\} \subseteq T, \{x',y'\} \subseteq T, \{x,y\} \neq \{x',y'\}\}$

Now we no longer need the definition, so we discard (5) from the formulation.

Next, we need a general refinement rule:

> To refine a set variable of fixed cardinality n drawn from a set A of size m, totally ordered by $\leq$, introduce a set S of n decision variables, $\{s_1, \ldots, s_n\}$. The domain of each s_i is A. Break symmetry among the s_i variables by adding the constraint $s_1 < s_2 < \cdots < s_n$. We could also build into this the simple bounds consistency argument by taking the domain of each s_i to be $\{A_i, \ldots, A_{m-n+i}\}$, where A_i is the i^{th} element in the ordering of the elements of A.

Applying this rule results in the following problem formulation:

10. Given n, find $S = \{s_1, s_2, \ldots, s_n\}$,
    where each s_i has domain $\{i-1, \ldots, n^2-n+i-1\}$.
11. $s_1 < s_2 < \cdots < s_n$
12. Minimise($max(S)$)
13. $\{|x-y| \neq |x'-y'| \mid \{x,y\} \subseteq S, \{x',y'\} \subseteq S, \{x,y\} \neq \{x',y'\}\}$

In the case of our example, a worthwhile transformation is to introduce a set of distance variables, $d_{\{x,y\}}$.

14. $\{d_{\{x,y\}} = |x-y| \mid \{x,y\} \subseteq S\}$

This is a lifted form of the introduce rule of Section 4. We can now perform a lifted version of eliminate; substituting (14) into (13) gives:

15. $\{d_{\{x,y\}} \neq d_{\{x',y'\}} \mid \{x,y\} \subseteq S, \{x',y'\} \subseteq S, \{x,y\} \neq \{x',y'\}\}$

It should not be difficult to notice that (15) defines a clique. A lifted version of our all-different introduction rule would replace (15) with:

16. all-diff($\{d_{\{x,y\}} | \{x,y\} \subseteq S\}$)

At this level of abstraction we now use symmetry breaking constraint (11) to simplify (12) to:

17. Minimise(s_n)

The set of all variables x and y such that $\{x,y\} \subseteq S$ is equivalent to the set of pairs of variables x and y such that $x \prec y$ and $x, y \in S$, where $\prec$ is an arbitrary total ordering of S. In this particular case, we choose $\prec$ to be defined as $s_i \prec s_j \leftrightarrow i < j$. Below we shall see the significance of this choice. Hence, (14) and (16) can be refined to:

18. all-diff($\{d_{s_i,s_j} | s_i, s_j \in S \text{ and } i < j\}$)
19. $\{d_{s_i,s_j} = |s_i - s_j| \mid s_i, s_j \in S \text{ and } i < j\}$

Now, using (11), (19) can be simplified to

20. $\{d_{s_i s_j} = s_j - s_i \mid s_i, s_j \in S \text{ and } i < j\}$

The final problem formulation is as follows.

- Given n find S=$\{s_1, s_2, \ldots, s_n\}$,
  where each s_i has domain $\{i-1, \ldots, n^2 - n + i - 1\}$.
- $s_1 < s_2 < \cdots < s_n$
- Minimise(s_n)
- all-diff($\{d_{s_i,s_j} | s_i, s_j \in S \text{ and } i < j\}$)
- $\{d_{s_i,s_j} = s_j - s_i \mid s_i, s_j \in S \text{ and } i < j\}$

With n taken as three, it is equivalent to the final representation in Table 5. Symmetry amongst the tick variables has been broken, and all-different difference variables for the inter-tick distances have been introduced.

Working with a more abstract input language that uses sets allows us to avoid the problem of detecting symmetry. We also retain the advantage of making valid transformations for the whole problem class. Furthermore we avoid the overwhelming size of the input that inevitably causes slowdown when transforming naive representations of individual instances. Refinement rules allow us to move to progressively more concrete levels of abstraction as necessary to perform transformations at the appropriate level.

6 Conclusions

We have described CGRASS, a system for the automatic transformation of a naive model of a constraint satisfaction problem into one that requires significantly less effort to solve. CGRASS adopts a rule-based architecture, transforming a model via the application of rules which encapsulate modelling expertise. The set of rules described here should be viewed as a representative sample. It is not complete in any sense, and we will continue to extend it in future.

The current implementation of CGRASS is able to transform individual instances only. Results obtained from experiments on the Golomb ruler problem suggest that this approach is impractical in general. We have discussed the need for the ability to reformulate problem classes, both to avoid dealing with large instances and in order that the inferences made hold for every instance of the class. We have also argued that it is important to reason at higher levels of abstraction. Refinement rules allow the systematic creation of effective concrete models from abstract formulations, and we conjecture that transformations at higher levels of abstraction will sometimes prove easier than at more concrete levels. As an example, we have outlined how the Golomb ruler problem might be effectively transformed from a high level description into a good model. Space prevents us from giving further examples here, but elsewhere [7] we have applied a similar process to the SONET network design problem.

A principal element of future work is to extend CGRASS' set of rules to allow it to reason about parameterised problem classes and at higher levels of abstraction. In particular, we will introduce refinement rules to move from higher to lower levels of abstraction as necessary. We will also consider whether CGRASS' rules can be implemented via constraint handling rules [9]. Possible obstacles include more complex pattern matching, the use of best-first search as the rule base grows in complexity, and the use of an executive which decides when to stop inferring and start searching.

Acknowledgements This project is supported by EPSRC Grant GR/N16129[2]. The third author is supported by Science Foundation Ireland. We thank Brahim Hnich for discussions about refinement. Julian Richardson's adaptation of PRESS to manipulate inequalities and its success at automatically generating a number of implied constraints influenced our approach. Finally we thank our anonymous referees for their useful comments.

References

1. R. Boppana and M. M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32:180–196, 1992.
2. A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.
3. S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.
4. S. Colton and I. Miguel. Constraint generation via automated theory formation. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 575–579, 2001.
5. A.K. Dewdney. Computer recreations. *Scientific American*, pages 16–20, December 1985.
6. M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.
7. A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, and T. Walsh. Towards model reformulation at multiple levels of abstraction. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*, pages 42–56, 2002.
8. A.M. Frisch, I. Miguel, and T. Walsh. Extensions to proof planning for generating implied constraints. In *Proceedings of Calculemus-01*, pages 130–141, 2001.
9. T. Frühwirth. Theory and practice of constraint handling rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, volume 37(1–3), pages 95–138, 1998.
10. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proof. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence 624*. Springer-Verlag, 1992. Also available as Research Report 592, Dept of AI, Edinburgh University.
11. G.F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-wesley, 1998.
12. S Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
13. J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on AI*, pages 362–367. American Association for Artificial Intelligence, 1994.
14. B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the 16th National Conference on AI*, pages 182–187. AAAI, 2000.
15. B.M. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb ruler problem. In *Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints*. International Joint Conference on Artificial Intelligence, 1999.
16. P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

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