

# Automated Reformulation of Constraint Satisfaction Problems

John Charnley\*  
jwc04@doc.ic.ac.uk

Simon Colton\*  
sgc@doc.ic.ac.uk

Ian Miguel†  
ianm@dcs.st-and.ac.uk

\* Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2AZ

† School of Computer Science, The University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SX

## 1 Overview

A constraint satisfaction problem (CSP) consists of a set of variables  $\{x_1, x_2, \dots, x_n\}$ , a set of domains of values the variables can take and a set of constraints specifying which values the variables can take simultaneously. A solution to a CSP is an assignment of values to each of the variables from their domains such that no constraints are broken. They find widespread use in science and industry and can be extremely complicated, involving a large number of variables and complex constraints. CSP solvers allow users to specify CSPs in a particular syntax, and then search for solutions to the problem, normally using a configurable search approach. Correctly formulating CSPs is a skilled and time-consuming task. Moreover, once formulated, a CSP problem can take a large amount of processing time to solve. For these reasons, various methods have been devised to improve the effectiveness of CSP solving. One such method is to find additional information about the domain being studied and use this knowledge to reformulate the CSP solver to improve its effectiveness. In particular, when the domain of investigation is described by axioms in first-order logic then it may be possible to derive new theorems from those axioms. Such *implied* theorems are true for all instances of the domain and can therefore be added to the CSP formulation without loss of generality.

Colton and Miguel (2001) used a machine learning system to discover new information about the problem domain of certain CSPs. Their approach was semi-automatic and significant aspects were performed manually. In particular, a skilled constraints programmer created the initial solver models and translated new information into constraints and a skilled mathematician manually reviewed the learning system output. Consequently, although the reformulations provided an improvement in the speed of finding solutions, the gain was offset by the large amount time and expertise required to pre-process the problem.

We have developed a system that fully automates this approach. The system consists of a number of modules, one of which is the HR machine learning system described in Colton (2001). HR combines machine learning and automated theorem proving to generate concepts and proven conjectures for a given domain. In the mode of operation that we use, HR starts with the basic axioms of the domain, some examples of objects which satisfy those axioms and some basic concepts of that domain. New concepts are discovered by passing existing concepts through a number of production rules. HR empirically checks the sets of examples which satisfy concept definitions and formulates conjectures whenever there is some relationship between the example sets. HR uses the Otter theorem prover, described in McCune (1990), to prove conjectures. Our system also uses the Constraint Logic Programming in Finite Domains library (CLPFD), described in Carlsson et al. (1997), which is part of the SICStus Prolog programming suite. The representation language used by the system is first-order logic, which is understood by HR and Otter. However, CLPFD has its own syntax and so a key part of our system is a bespoke translation suite to convert first-order logic sentences and concepts into CLPFD syntax.

The system is suited to reformulating CSPs for families of related problems with the same axioms expressible in first-order logic e.g. finite algebras. The starting point is the axioms for the family of problems and a template for generating solver models for that domain. The user has two options in creating an initial solver model. They can program the model into the domain template, requiring some knowledge of constraint programming, or they can have the system generate a starting model by automatically translating the domain axioms, using the automated translation routines. In some cases, a combination of both approaches is used. The base solver is used to find solutions to small problem instances, which are used, together with the axioms of the domain, to configure HR for a discovery run.

The proven conjectures HR finds are translated into CLPFD syntax constraints that can be added seamlessly into a solver model, as outlined in 2. We apply a generate-and-test approach to finding the best reformulation i.e. the solver model that finds all solutions to the given problem in the shortest time. This will be the base model plus zero or more of the theorems HR has discovered. Individual theorems are assessed by considering their impact upon effectiveness when added to the base model. Testing starts with an initial population of all the theorems discovered by HR. Successive rounds consider combinations of increasing numbers of the best theorems from previous rounds. Testing ends when no improvement can be found, or a user-defined limit is reached, whereupon all reformulations from all rounds are ranked.

## 2 Interpretation and Translation of Constraints

First-order logic sentences to be translated into CLPFD syntax are first parsed into nested prolog terms using a Definite Clause Grammar. The translator then recurses through the nested expression creating, in most instances, a new predicate for each level of nesting. We employ methods such as *flattening* and *case-splitting* to improve constraint efficiency. *Flattening* refers to carefully re-translating theorems to simplify them without changing their meaning, for example converting  $a * b = c \wedge b * a = c$  to  $a * b = b * a$ . *Case-splitting* allows us to post simpler, more targeted, constraints representing implication conjectures. For instance, if we take the contra-positive to the conjecture  $\forall a b ((a * b = b * a) \rightarrow (a = b))$ , then we can case-split between  $a = b$  and  $a \neq b$ , posting the constraint  $a * b \neq b * a$  in the latter case.

The main method for translating quantifiers creates predicates to generate sets of value combinations for the quantified variables. Further predicates are used to enforce that a sub-expression holds for all combinations, in the case of universal quantifiers, and for at least one combination, in the case of existential quantifiers. Another method we have used for existential quantification is to represent the quantified variables as additional solution variables and add them to the solver. The solver must assign values to these new variables when finding a solution, which improves efficiency.

Equality expressions require recursive translation, where each sub-expression is first translated in such a way as to produce a result, which are then constrained to be equal. Conjunctions, disjunctions and negations are all translated into the appropriate respective CLPFD syntax in a relatively straightforward manner. Depending upon the domain, some expressions represent variables the solver is trying to allocate values to. For example, in finite algebras, the results of operators, e.g. expressions of the form  $a * b$ , are the solution variables. If  $a$  and  $b$  are known then we can identify exact solution variable affected by the expression. For example,  $\forall a b (a * b = b)$  allows us to constrain all variables  $(a * b)$  explicitly prior to search. Where nesting occurs, e.g. in  $(a * b) * c$ , the affected solution variables are only determined when values have been assigned to variables during search. These are translated using the built-in *element* constraint, which is optimised to resolves list values and positions at search time. CLPFD syntax requires that some expressions, e.g. implications, be *reified*, i.e. represented by boolean variables. In the syntax, sub-expressions of reified expressions must also be reified. Thus two translation methods have been developed, with and without reification.

## 3 Results

We have used the system to investigate the area of finite algebras. We started with algebras with one operator ( $*$ ), e.g. quasigroups, and have extended it to algebras with multiple operators e.g. rings. In many cases, the system automatically produced a reformulation that significantly outperformed the model based upon the starting axioms alone. In particular, the system found and used all the axioms discovered in Colton and Miguel (2001) using the semi-automatic approach. For example, in the case of QG3 quasigroups, our system automatically reformulated the base model to include the axiom  $\forall b c d (((b * c = d \wedge c * b = d) \rightarrow (b = c)))$ , reducing the time taken by the solver to find examples of size 8 by 67.9%, some 7 hours. In addition, full automation of assessment and translation means there is no longer a requirement for the users to be particularly skilled in either constraint programming or the domain of investigation. Further, we note that all the useful theorems represent mathematical properties that are interesting in their own right. Consequently, the system is useful in assessing the interestingness of mathematical output from a machine learning system.

There were a number of algebras where the system did not find an improvement upon the base solver. There are several possible reasons for this. Firstly, it could be that our translation process isn't specifying the most computationally efficient constraints. Alternatively, we may not be detecting useful constraints due to inadequacies of our testing approach, or we may not be finding useful theorems due to the time or configuration of HR. It could simply be the case that no useful derived theorems would be found regardless of how much we looked, in which case the approach could never succeed. Each of these areas can be considered for further work. The system is currently limited to first-order logic, which is insufficiently expressive to cover all CSP problems. We would like to generalise our approach accordingly.

## References

- M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.
- S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag UK, 2001.
- S. Colton and I. Miguel. Constraint generation via automated theory formation. In *Proceedings of the Seventh International Conference on the Principles and Practice of Constraint Programming*, Cyprus, 2001.
- W. McCune. The Otter user's guide. Technical report, ANL, 1990.