

Automatic Generation of Implied Constraints

John Charnley, Simon Colton¹ and Ian Miguel²

Abstract. A well-known difficulty with solving Constraint Satisfaction Problems (CSPs) is that, while one formulation of a CSP may enable a solver to solve it quickly, a different formulation may take prohibitively long to solve. We demonstrate a system for automatically reformulating CSP solver models by combining the capabilities of machine learning and automated theorem proving with CSP systems. Our system is given a basic CSP formulation and outputs a set of reformulations, each of which includes additional constraints. The additional constraints are generated through a machine learning process and are proven to follow from the basic formulation by a theorem prover. Experimenting with benchmark problem classes from finite algebras, we show how the time invested in reformulation is often recovered many times over when searching for solutions to more difficult problems from the problem class.

1 Introduction

Constraint programming is a successful technology for tackling a wide variety of combinatorial problems in disparate fields, such as scheduling, industrial design, and combinatorial mathematics [18]. To use constraint technology to solve a constraint satisfaction problem (CSP), its solutions must first be characterised, or *modelled* by a set of constraints on a set of decision variables. Constraints provide a rich language, so typically many alternative models exist for a given problem, some of which are more effective than others. Formulating an effective constraint program is a challenging task, for which new users lack expertise. Great value is placed, therefore, on any method that helps to reduce this modelling bottleneck.

One method of improving a model is to introduce constraints implied by the model. We describe here a method for introducing such *implied constraints*. As described in §1.1, this fully automated method is a significant improvement upon the semi-automated method presented in [6]. Given a set of problems from the same problem class, as detailed in §2, we employ a machine learning module to induce relationships in the solutions for small problems and we use a theorem prover to show that the truth of the relationships follow from the CSP model. As described in §3, the relationships are translated and interpreted as implied constraints which are added to the CSP model. On larger instances of the problem class, the additional constraints often lead to a much reduced solving time. In §4, we present the results of applying this approach to some benchmark CSPs, namely finite algebras. We show how the system was able to improve upon the basic solver model in a number of cases and the reformulation time was soon recovered when searching for larger examples. For instance, in one case, we were able to reduce the solving time to 16% of the original, saving around 23 hours of processing.

1.1 Background

In [6], Colton and Miguel describe the application of machine learning and theorem proving to the generation of implied constraints. They employed descriptive machine learning to generate hypotheses about concepts expressed in the basic model of a class of CSPs. They then used a theorem prover to show that certain hypotheses followed from the CSP model, and hence they could be added as implied constraints. While they achieved significant increases in efficiency in this manner, their approach was semi-automatic. In particular, both Miguel and Colton, with expertise in constraint modelling and the domain of application (pure mathematics) respectively, were required to interpret the output from the learning system as constraints and translate them into the language of the constraint solver. Consequently, although the reformulations provided an improvement in solver efficiency, this was offset by the large amount of time and expertise required during the reformulation process.

As described in the next section, our approach makes use of a number of AI techniques to fully automate the process that Colton and Miguel introduced. For brevity, we assume general familiarity with automated theorem proving [7], constraint solving [14] and machine learning [16]. We combine techniques from these areas via usage of the following systems:

- The Otter automated theorem prover, which uses the resolution method to prove theorems by refutation [15].
- The Constraint Logic Programming in Finite Domains (CLPFD) library [4] included with SICStus Prolog.
- The HR automated theory formation system. This performs descriptive learning to speculatively invent concepts and form hypotheses in a domain of interest. HR is extensively described in [5].

Our approach is distinct from, and complementary to, existing automated constraint reformulation methods. For example, the CGRASS system [9] captures common patterns in the hand-transformation of constraint satisfaction problems in transformation rules. Given an input model, CGRASS applies the rules in a forward chaining manner to improve the model gradually. A key difference from the approach presented here is that CGRASS is applicable to single problem instances only, rather than problem classes. The O'CASEY system [12, 13] employs case-based reasoning to capture constraint modelling expertise. Each case records a problem/model pair. Given a new problem, the most similar problem is retrieved from the case base. A model for the new problem is then constructed by following the modelling steps used in the retrieved case. Bessiere et al. present a method for learning the parameters of certain types of implied constraint by considering solutions and non-solutions of relaxations of the original problem [3]. The CONACQ system [1, 2] uses a version space approach to learn a CSP model from scratch given only positive and negative example solutions to the CSP.

¹ Department of Computing, Imperial College, London, United Kingdom

² School of Computer Science, University of St Andrews, United Kingdom
email: jwc04@doc.ic.ac.uk, sgc@doc.ic.ac.uk and ianm@cs.st-and.ac.uk

2 The Reformulation Process

A CSP *problem class* is a possibly infinite set, $\{P_1, P_2, \dots\}$ of constraint satisfaction problems. *Instances* of the class are obtained by giving values for the *parameters* of the class. We focus on classes parameterised by a single positive integer n . The number of variables in P_n increases with n , making successive members of the class increasingly difficult to solve. However, the core constraints for each P_i can be expressed using the same general relationships over the variables. We have been working with CSPs where each variable has the same domain. For problem instance P_n from a class of CSPs, this domain is $\{1, \dots, n\}$. Hence, in using the term *domain size*, we mean the size of the domain of any variable, namely n for P_n . A good example of such a class of CSPs is Latin squares, where P_n represents the problem of filling in an n by n grid with the integers 1 to n such that no number appears twice in any row or column.

Our aim is to take the core model of a problem class and automatically produce a reformulated core model containing additional constraints. The additional constraints will be proven to be implied by the core model, so that a solution to any P_n is also a solution to the reformulated version of P_n , and vice versa. We have automated the following approach to reformulating CSPs:

1. The user supplies information about a problem class, including the core constraints. We call this formulation of the CSP the *basic model*.
2. The CLPFD solver generates solutions to small problem instances.
3. The solutions are given, along with the core model, to the HR system. This generates empirically true conjectures which relate concepts expressed in the core constraints.
4. Each conjecture is passed to Otter, and an attempt is made to show that the conjecture is implied by the core model.
5. Each proved conjecture is interpreted as an implied constraint and translated into the syntax of the CLPFD solver. As described in §3, automating this aspect of the process was the most challenging.
6. Each implied constraint is added to the basic model to produce a reformulated model. The small problem instances are then solved again using the reformulated model. Every implied constraint used in a reformulation which improves – in terms of efficiency – upon the basic model, is recorded in a set E .
7. Every pair of constraints from E are added to the basic model and tested for an efficiency increase. Every triple of constraints is then tested, etc., until a user-defined limit is reached, or no reformulation improves upon the best model so far in a particular round.
8. All reformulations leading to a speed up for the small problem instances are presented to the user in decreasing order of the efficiency gained from the reformulation.

As an example, we started with the class of QG3 quasigroup problems (defined in §4). We formed the basic model for this from the first order axioms of QG3 quasigroups, using the translator in our system. Using Sicstus to solve small problems from this class, our system generated 8 QG3 examples. HR used these to discover 100 conjectures about QG3 quasigroups, which were proved by Otter. The first round of testing highlighted: (i) $\forall bcd ((b * b = c \wedge d * d = c) \rightarrow (b = d))$ and (ii) $\forall bcd ((b * c = d \wedge c * b = d) \rightarrow (b = c))$ as theorems which – when interpreted as constraints – improve the efficiency of the basic model. The constraints arising from these theorems were further combined into a single model in the second round. In total, the reformulation process took approximately 8 minutes, and produced a much improved model (see §4).

3 Constraint Interpretation

The three AI systems we have been working with are all able to work with first order representations. We note that CSP is in NP, and, from Fagin’s theorem [8], NP is the set of languages expressible via existential second-order logic. However, in order to provide an initial benefit via commonality of language, at present, we have limited ourselves to working with CSPs that can be expressed in first order logic.

The HR system has been developed to work with Otter-style first order representations, so the major difficulty of translation was in terms of converting conjectures from Otter syntax to CLPFD constraints. Given a proven conjecture from HR expressed as a string in Otter-style first order syntax, the translation process first tokenizes the string and then parses it using a bespoke Definite Clause Grammar (DCG). For example, the first-order string:

$$all\ a\ b\ (exists\ c\ d\ (a * c = b \ \&\ d * a = b))$$

would be parsed as the following expression:

$$all([var(a), var(b)], exists([var(c), var(d)], e(\&, e(=, e(*, var(a), var(c)), var(b)), e(=, e(*, var(d), var(a)), var(b)))))$$

Once parsed, expressions are translated into constraints in CLPFD syntax. This is achieved via recursion through the nested expression which creates, in most instances, a new predicate for each level of nesting. How the interpretation method deals with quantifiers, expressions and solution variables is described in §3.1 and §3.2, below. To improve the computational efficiency of the implied constraints, further processing via simplification and further interpretation via contraposition and case splitting is undertaken, as described in §3.3.

3.1 Quantifiers

The DCG produces partially translated universal quantifiers in the form $all(variable_list, sub_expression)$. This states that $sub_expression$ should hold for each combination of the values of the variables in $variable_list$. This is further translated by introducing three new predicates. The first predicate creates a list of all possible value combinations (pairs, triples, etc.) as a list of sets determined by the domain size. The second predicate recurses through this list, and makes a call to the third predicate for each value combination. The third predicate represents $sub_expression$ and is constructed accordingly. The next recursion level is passed details of how the first order variables have so far been translated into Prolog variables, so they can be correctly matched.

As an example, the translation of:

$$all([var(a), var(b)], e(=, var(a), var(b)))$$

produces the following triple of logic programs:

$$\begin{array}{lll} A_1(N):- & A_2([]). & A_3(V1,V2):- \\ \text{sub_sets}(N,2,S), & A_2([V1,V2][Ss]):- & V1 \# = V2. \\ A_2(S). & A_3(V1,V2), & \\ & A_2(Ss). & \end{array}$$

Here, N is the domain size and S is a list of value combinations (in this case, pairs) for $var(a)$ and $var(b)$, which is produced by the predefined `sub_sets` predicate. Predicate `A_1` creates this list and passes it to `A_2`, which recursively calls `A_3` for each combination.

Existential quantifiers are partially translated to the form $exists(variable_list, sub_expression)$, which states that

sub_expression holds for at least one combination of the values of the variables in *variable_list*. We use two methods to translate existential quantifiers. One method creates all combinations of variable values as before and, using the solver's disjunction syntax, produces another predicate which constrains that at least one case should hold. The second method, which is more effective where it can be used, involves representing the existentially quantified variables as new solution variables in the CSP. These new variables are given domains appropriate to the size of the problem and are added to the list of required solution variables. Of course, in finding a solution, the solver must ensure that these variables are given a value, which enforces their existence.

As an example, using the second method with the following partially translated sentence:

$$exists([var(a), var(b)], e(=, var(a), var(b)))$$

produces the following pair of predicates:

```
B_1(N,OldVarList,NewVarList):-
  domain(V1,1,N),
  domain(V2,1,N),
  B_2(V1,V2),
  append(OldVarList,[V1,V2],NewVarList).

B_2(V1,V2):-
  V1 #= V2.
```

Here, predicate B_1 creates two new solution variables, V1 and V2, to enforce the existence of valid values for first order variables *a* and *b*. These values are passed to B_2, which constrains that their value be equal for whatever instantiation the solver chooses to give them.

3.2 Expressions and Solution Variables

Expressions are partially translated as: $e(op, -, -)$, for instance $e(\&, sub_exp_a, sub_exp_b)$ and $e(=, sub_exp_a, sub_exp_b)$. Depending upon the domain of investigation, particular expressions represent solution variables. For example, in single-operator finite algebras, the solution variables are the result of the algebraic operator on all pairs of elements. Consequently, any expression in the form $e(*, n, m)$, where *n* and *m* are ground integers, identifies a specific solution variable. In other domains, different expressions represent the solution variables, for instance the inverse operator in groups.

The majority of first-order expressions that HR produces can be translated such that individual solution variables are explicitly identified. In some instances, however, where operators are nested, it is not possible to identify the exact solution variables that are affected, as they may vary according to the parameters of the constraint. For example, consider the sentence $\forall a b ((a * b) * (b * a) = b)$. Here, we must first know the results of $a * b$ and $b * a$ before we can identify the solution variable whose value must be constrained equal to *b*. The translator identifies nesting and employs the CLPFD *element* constraint, which is common to many solver systems. This built-in constraint is optimised to resolve values within lists during search.

In general, a conjunction is translated using standard Prolog comma syntax. Equality expressions equate the value of two evaluable sub-expressions. For example, $\forall a b (a * b = b * a)$ equates the result of $a * b$ with that of $b * a$. When translating equalities, a new variable is generated to represent this equated result and each sub-expression is recursively translated in such a way that its result is this new variable. For example, $\forall a b (a * b = b * a)$ generates the following triple of predicates:

```
C_1(N):-
  sub_sets(N,2,S),
  C_2(S).

C_2([],_).
C_2([[V1,V2]|Ss]):-
  C_3(V1,V2),
  C_2(Ss).

C_3(V1,V2):-
  mv(V1,V2,V3),
  mv(V2,V1,V3).
```

Here, C_1 and C_2 handle universal quantification. Predicate C_3 calls the $mv(x,y,z)$ predicate which finds the solution variable representing $x*y$ and constrains it to be equal to *z*. Equality of the sub-expressions, $a * b$ and $b * a$, is enforced by having both calls to *mv* return V3.

CLPFD syntax requires some expressions, such as implications and negations, to be expressed using reification variables to represent the constraints for the sub-expressions. Reification variables, which take the value *true* if the constraint holds and *false* otherwise, allow the solver to efficiently propagate constraints during search and are attached to constraints using the CLPFD $\#<=>$ syntax. The sub-expressions of expressions requiring reification must also be reified. Consequently, the translator is able to translate all expressions both with or without reification as necessary.

3.3 Case-splitting and Simplification

The translation process can identify where conjectures can be interpreted as case-splits over quantified variable values. Case-splits allow us to consider specific sets of solution variables and can reduce the complexity of some conjectures by expressing them as simple and efficient constraints. Consider, for example, the theorem: $\forall a b ((a * b = b * a) \rightarrow (a = b))$, which states an anti-Abelian property (that no two distinct elements commute). The contra-position of this theorem is: $\forall a b ((a \neq b) \rightarrow (a * b \neq b * a))$, which can be interpreted as a case-split on the variables *a* and *b*, i.e., when they are equal and when they are not equal. Only when they are unequal do we post the inequality constraint.

The translator finds case-splits by considering whether or not the sub-expressions of an implication contain solution variables. If they do not, then the implication is translated as a case-split using the theorem or it's contra-position as appropriate. Case-split translations introduce an additional predicate which succeeds if the variable values meet the case-split requirements, in which case the constraints are posted, and fails otherwise. As an example, the translation of the above anti-Abelian property, after universal quantification, is shown below:

```
D_1(V1,V2):-
  D_2(V1,V2),
  D_3(V1,V2).

D_2(V1,V2):-
  V1 \= V2.

D_3(V1,V2):-
  mv(V1,V2,V3),
  mv(V2,V1,V4),
  V3 #\= V4.
```

In this example, D_1 represents $((a \neq b) \rightarrow (a * b \neq b * a))$. The predicate D_2 is used to determine the case, if *a* and *b* (V1 and V2) are unequal then it will succeed and a call to D_3 will be made, posting the constraint $(a * b \neq b * a)$. If it fails, i.e., $a = b$, then no constraints are posted.

In some cases, it is possible to remove variables or re-translate theorems without affecting meaning. In some cases, such simplification can improve the effectiveness of the constraints produced from the theorem. For example, $\forall a b c ((a * b = c \wedge b * a = c) \rightarrow (a = b))$ can be simplified to $\forall a b ((a * b = b * a) \rightarrow (a = b))$. Here we see that the variable *c* – which is simply a marker for equality – has been removed. Care is taken to avoid inappropriate simplification. For example, in the theorem: $\forall a b c ((b * c = c \wedge c * d = c) \rightarrow (b = d))$, we cannot simply remove the variable *c* without changing the semantics of the theorem.

Algebra	No. ex.	Test order	HR time	Test time	Total time	Proportion (%)
QG7	12	6	9:31	6:29	16:00	8.6
QG6	2	6	11:35	1:37	13:12	12.2
Group	22	5	3:31	21:00	24:32	63.1
QG3	8	5	5:32	1:55	7:28	68.8
QG4	20	5	4:29	2:28	6:56	70.4
Ring	25	5	9:17	19:48	29:06	70.6
QG5	8	5	12:57	4:15	17:13	71.7
Moufang	25	3	8:07	7:37	15:44	85.8
QG1	16	4	15:26	7:01	22:28	100.0
QG2	16	4	20:48	6:51	27:39	100.0
Loop	16	5	6:42	3:07	9:50	100.0
Medial	22	3	24:08	2:37	26:45	100.0
Semigroup	23	3	4:15	1:39	5:55	100.0
Monoid	32	4	4:32	4:55	9:28	100.0

Table 1. Reformulation times (minutes:seconds) and proportion of basic model solving time for training (small) domain sizes

4 Experiments and Results

Finite algebras provide first-order benchmark tests for CSP solvers. In particular, QG-quasigroup existence problems have driven much research in constraint solving and other computational techniques. QG-quasigroups are Latin squares with extra axioms, e.g., QG3 quasigroups are Latin squares for which $\forall a b ((a * b) * (b * a) = a)$. Further details are available at the CSPLib.³

Our experimental setup was as follows. We started with the axioms of a particular finite algebra, which effectively defines the core model of a CSP problem class. The axioms were supplied in Otter-style first order syntax and were translated into the basic model of a CSP using the translation process described above. The system ran the basic solver model for sizes up to n , and stopped when the time to solve for size $n + 1$ took over 2 minutes. In this context, and for all the experiments, by *solve* we mean exhausting the search space and finding all solutions. The solutions for the small sizes were passed to HR, which ran until it found 100 proven conjectures (with Otter providing the proofs).

The setup for HR was similar to that in [6]. However, we also supplied two additional background concepts, namely: the concept of two elements in the algebra being equal and two elements being unequal. We have found that this increases the yield of computationally effective constraints found by HR. In addition, we set up HR so that it used Otter to discard any conjectures which were tautologies, i.e., could be proved without the axioms of the domain. This slightly increased HR’s processing time, but greatly reduced the number of reformulations to test, hence improved overall efficiency.

The proved conjectures from HR were interpreted and translated as constraints and used to reformulate the CSP problem using the process described above. For any reformulation shown to improve efficiency on the small – training – problem instances, we compared performance against that of the basic model for problems of size $n+1$ and larger. Table 1 shows the number and size of small examples found during the reformulation stage, the time taken by HR, the time taken to construct good reformulations and the total reformulation time. This table is ordered by decreasing efficiency gain. Where we improved upon the basic model, table 2 shows the results of taking the best reformulation and applying it to larger problem sizes.

In 8 out of 14 cases, a gain in efficiency was achieved via the reformulated model. Unfortunately, there is a memory bound of 256 Mb. in the Sicstus solver, which effectively meant that we couldn’t

Algebra	Domain size	Basic model	Reformulated model	Proportion (%)
QG3	7	3:09	1:24	44.5
	8	10:07:02	3:10:03	31.3
QG4	6	0:07	0:04	54.2
	7	11:19	5:18	46.8
QG5	7	1:24	1:17	91.7
	8	38:05	28:52	75.8
QG6	9	27:25	6:25	23.4
	10	24:21:00	5:53:03	24.2
QG7	8	19:12	3:33	18.5
	9	27:12:35	4:19:42	15.9
Group	8	16:37	4:15	25.6
	9	4:36:39	28:27	10.3
Moufang	4	0:11	0:08	72.3
	5	10:49	4:19	39.9
Ring	7	0:37	0:30	79.8
	8	4:22	2:09	49.5

Table 2. Solving times (hours:minutes:seconds) and proportion of basic model solving time for testing (larger) domain sizes. Note that only those algebras where a speed-up was identified in table 1 are shown

perform tests for problem instances larger than those shown in table 2. In those cases where we did manage to run longer tests, the benefit of reformulation is clear. In particular, for QG3, QG6, QG7 quasigroups and groups, the time taken to produce the reformulated model was a fraction of the time gained in running the larger tests. In the other cases, it seems likely that the reformulation time would be similarly gained back at larger domain sizes.

In [6], Colton and Miguel studied reformulations for QG-quasigroup problems, which were generated in a partially automated, partially hand-crafted way. It is interesting to note that the fully automated method described here re-created all the best reformulations found by hand in [6], with two exceptions. The first exception was with the reformulations of QG3 and QG4 quasigroups. Here, the best fully automated reformulation for both QG3 and QG4 uses the constraints generated from this pair of theorems:

$$\begin{aligned} \forall bcd ((b * c = d \wedge c * b = d) \rightarrow (b = c)) \\ \forall bcd ((b * b = d \wedge c * c = d) \rightarrow (b = c)) \end{aligned}$$

These state, respectively, that these quasigroups are anti-Abelian (i.e., no two distinct elements commute) and that the main diagonal elements must all be distinct. As described in §3.3, the system uses simplification and the contra-positive of each of these to post constraints for all cases where $b \neq c$. To illustrate these constraints, at size 6, the first of these theorems results in 15 inequality constraints of the form $X_{ij} \neq X_{ji}$ for $1 \leq i, j \leq 6$ and $i \neq j$. In [6], however, their best reformulations also used another theorem: $\forall a b (a * a = b \rightarrow b * b = a)$. In our experiments, this theorem was found in some reformulations for QG3 and QG4 which improved efficiency. However, these reformulations were less efficient than the best reformulation.

The second exception represents one of numerous instances where the system discovered implied constraints which were not identified in [6]. In particular, the best reformulation for QG5 quasigroups from [6] was identified as the second best reformulation in our experiments. The best fully automatic reformulation used constraints derived from this new theorem:

$$\forall bcd ((b * c = d \wedge d * b = c) \rightarrow (c = d))$$

For QG5 quasigroups up to size five, on average, the solving time for the second best reformulation (i.e., the reformulation identified in [6]) took 77% of the time taken using the basic model. With the constraints derived from the above theorem, this time reduced to 71.7% that of the basic model.

³ A library of CSP benchmark problems (www.csplib.org).

5 Conclusions and Further Work

We have developed an automatic system, which uses machine learning and automated theorem proving to generate implied constraints for CSPs, thus improving the effectiveness of a CSP solver by reformulating the model of the problem. Using finite algebras as benchmark tests, in many cases, the time taken to reformulate problems represents a worthwhile investment, which is soon recovered when searching for solutions to larger problem instances. We have shown clear progress over the approach presented in [6] by (a) automating their semi-automated approach – which required much domain knowledge and constraint solving expertise – while recreating similar or better efficiency gains (b) successfully applying the procedure to new finite algebras, namely groups, Moufang loops and rings, the last of these being exceptional in that they have two algebraic operators rather than one, and (c) discovering new and effective implied constraints not identified in [6].

There is still room for improvement. In particular, in some cases, the system failed to create an improved solver model. There are a number of possible reasons for this, including:

- The basic model is the best possible. In this case, the approach would never improve the basic model.
- Improving constraints do exist but were not found by HR. We could counter this by running HR for longer or with heuristic guidance specifically tailored to this process. In addition, we plan to implement new ways for HR to make conjectures, for instance by allowing the user greater control over the concept formation process [17], or by directly noticing all-different properties, for which there are efficient solving methods.
- Improving constraints were found by HR but translated inefficiently. We have endeavoured to make the translation process as effective as possible, but we can improve this process. In particular, where possible, we aim to interpret conjectures as all-different constraints rather than sets of inequalities.
- Our method of assessing the effectiveness of constraints fails to identify the most useful constraints. In particular, a constraint may not be effective at low order but its effectiveness may increase as the domain size increases. In contrast, a constraint may be highly effective for small test sizes, but degrade as the domain size increases. For instance, the reformulation for QG6 quasigroups used the idempotency constraint ($\forall b (b * b = b)$), which applies to elements on the main diagonal of a multiplication table. As we see from tables 1 and 2, this achieves a time of around 12% of that for the basic model for problem sizes up to 6, but around 24% for sizes 9 and 10. Hence, we plan to implement techniques for extrapolating the effectiveness of constraints in order to choose them more intelligently.

We have recently extended the translation capabilities to enable the formulation of some first-order theorem proving problems as CSPs. We also plan to generalise this approach further by enabling translation to the syntax of other solvers and extending into higher order logics. It is possible to use empirically plausible – but not necessarily proved – conjectures to define a case-split where two CSPs have to be solved in order to exhaust the search space, and we intend to pursue this possibility. Not only does this open up the possibility of distributing the solving process, but the sum of the times taken for the two cases may be less than the time taken to solve the basic model. Moreover, more complex case-splitting schemes can be used, which have already shown much promise in improving automated theorem proving techniques [11]. We also plan to study the ways in which concepts from HR can be used to streamline [10] constraint solving.

Here, the aim is to find a single solution, rather than all solutions, in which case generality can be sacrificed for efficiency by specialising a CSP. The ability to translate first order constraint satisfaction problems is a useful tool outside of the application to CSP reformulation. We intend to use this for various applications, such as the comparison of model generators and CSP solvers for benchmark tests.

The study presented here adds weight to the argument that combining reasoning techniques can produce AI systems which are more than a sum of their parts. For instance, due to the limited number of examples at small domain sizes, the reformulations of QG6 quasigroups were based on conjectures found by HR using only two examples. Such little empirical evidence probably wouldn't be statistically significant in a straight machine learning exercise, and hence the conjectures might be ignored. However, because they were proved by Otter, they were as valid as ones with greater empirical evidence, which highlights the advantages to be gained by combining different reasoning approaches.

Acknowledgements

Ian Miguel is funded by a Royal Academy of Engineering / EPSRC research fellowship. We would like to thank the anonymous reviewers for their helpful comments.

REFERENCES

- [1] C. Bessiere, R. Coletta, E.C. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automatic constraint acquisition. In *Proceedings of CP-2004*, pages 123–137, 2004.
- [2] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of ECML'05*, pages 23–34, 2005.
- [3] C. Bessiere, R. Coletta, and T. Petit. Acquiring parameters of implied global constraints. In *Proceedings of CP-2005*, pages 747–751. Springer, 2005.
- [4] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.
- [5] S Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.
- [6] S Colton and I Miguel. Constraint generation via automated theory formation. In *Proceedings of CP-2001*, pages 575–579, 2001.
- [7] D. Duffy. *Principles of Automated Theorem Proving*. Wiley, 1991.
- [8] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation, SIAM-AMS Proceedings 7, R Karp, ed. pages 43-73*, 1974.
- [9] A.M. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Proceedings of the Workshop on Constraint Solving and Constraint Logic Programming (LNAI 2627)*, pages 15–30, 2002.
- [10] C. Gomes and M. Sellman. Streamlined constraint reasoning. In *Proceedings of CP-2004*, 2004.
- [11] F Hoermann. Machine learning case splits for theorem proving. Master's thesis, Dept of Computing, Imperial College, London, 2005.
- [12] J. Little, C. Gebruers, D. Bridge, and E. Freuder. Capturing constraint programming experience: A case-based approach. In A. Frisch, editor, *International Workshop on Reformulating Constraint Satisfaction Problems at CP-2002*, 2002.
- [13] J Little, C Gebruers, D Bridge, and E Freuder. Using case-based reasoning to write constraint programs. In F. Rossi, editor, *Principles and Practice of Constraint Programming*. Springer, 2003.
- [14] K. Marriott and P. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
- [15] W McCune. The OTTER user's guide. Technical Report ANL/90/9, Argonne National Laboratories, 1990.
- [16] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [17] Pedro Torres and Simon Colton. Applying model generation to concept formation. In *Automated Reasoning Workshop*, 2006.
- [18] M. Wallace. Practical applications of constraining programming. *Constraints*, 1(1/2):139–168, 1996.