# Expressing General Problems as CSPs

John Charnley and Simon Colton

Department of Computing, Imperial College, 180 Queens Gate, London, SW7 2AZ.
email: jwc04@doc.ic.ac.uk, sgc@doc.ic.ac.uk

**Abstract.** We consider the translation of general AI problems into CSPs. In particular, we have developed a translation suite able to translate first order specifications into the syntax of the Sicstus CLPFD constraint solver. We describe recent extensions to the capabilities of this suite which have enabled it to handle problems outside of the algebraic domains for which it was designed. We demonstrate two of many advantages to having such a translation suite. Firstly, we show that an ability to translate between the syntaxes of different AI problem solving systems enables us to make meaningful comparisons of different AI techniques, and we demonstrate this using a model generator and a constraint solver on quasigroup problems. Secondly, with an ability to express a problem in different ways, we can begin to simulate more sophisticated problem solving which uses inductive, deductive and constraint solving techniques. We explore such possibilities with some applications to investigative reasoning, where the aim is to identify the cause of a phenomenon from a set of candidates.

## 1 Introduction

There is little fluidity in the standard AI paradigm, where an intelligent task is expressed as a problem to be solved and a particular technique (theorem proving, constraint solving, machine learning, planning, etc.) is chosen and stuck with. We believe that if we are to model more sophisticated problem solving techniques, then we will need to combine deductive, inductive, constraint solving and other methods. In order to do this, it is important to be able to translate between the syntaxes of different AI systems. We have developed a translation suite able to turn problem specifications in first order logic into the syntax of the Sicstus CLPFD constraint solver. We originally developed the suite to enable the use of first order machine learning and theorem proving techniques to introduce implied constraints in order to reformulate algebraic CSPs, as described briefly in §2 and fully in [2]. We have recently extended the suite to handle problem specifications outside of algebraic domains, as described in §3.

We demonstrate the utility of the translation suite in two ways (which can be taken in addition to the ability to reformulate CSPs described in [2]). Firstly, in §4 we show that meaningful comparisons of AI systems can be undertaken, and we demonstrate this with a comparison of the MACE2 and MACE4 model generators [6] [7] with the CLPFD solver. Secondly, in §5 we translate some puzzle problems from the TPTP library of first order theorem proving problems [9] and solve them with the CLPFD solver. These problems involve identifying a guilty suspect from a set of candidates. We discuss how combining reasoning techniques has much promise for such investigation problems, especially when there is insufficient background information available to solve the problem using stand-alone AI techniques.

## 2 Translation in Algebraic Domains

The translation suite was originally developed as part of the system described in [2]. In that system, first order theorems, in the syntax of the Otter theorem prover [5], were automatically translated into constraints for the SICStus CLPFD library [1]. Moreover, the system automatically reformulates CSP models for a given problem family by employing a machine learning system [3] to discover and prove (using Otter) theorems about the domain, which are implied by the axioms of that domain. These are then translated into constraints and added to the solver model to give a reformulated model with potentially better performance. The translation suite was used in both translating new theorems and in creating initial solver models from the domain axioms.

The first application of that system and, consequently, the original application of the translation suite, was in the domain of finite algebras. One family of finite algebras we studied was quasigroups, which are Latin squares with additional constraints, and are commonly used as a benchmark test for constraint solvers. A quasigroup of size $n$ is modelled as an $n * n$ square of integers of domain $\{1 \ldots n\}$, representing the result of $a * b,\ \forall a, b \in Q$, with the Latin square property interpreted as all-different constraints on the rows and columns of that square. Additional axioms for a particular subtype place more constraints upon the possible results of the operators, e.g., quasigroups of subtype QG3 have the additional constraint that: $\forall\, bc\, (b * c) * (c * b) = b$.

The formulae in these domains are composed of a number of common elements, several of which are present in the following formula:

$$\forall x\ \exists y\ (\neg(x * x = x) \rightarrow (y * y = y))$$

Here, we see both universal and existential quantification, two types of logical operation (implication and negation), and mathematical operators for equality and multiplication of elements.

The translation suite automatically creates the predicates representing the constraints by recursing through formulae one operator at a time. Full details of how each type of formula is processed is given in [2]. In brief, universal quantification is dealt with by generating wrapper predicates to consider all possible combinations. A similar approach is used to deal with existential quantification and this was supplemented by another efficient approach which considers existentially quantified variables as additional solution variables. The suite identifies those elements of formulae that represent the solution variables, i.e., the results of operators, and takes appropriate measures to deal with nesting of operators. Several techniques were developed to produce more efficient constraints. For example, unnecessary variables are removed in a process known as flattening and, where possible, certain formulae are identified as case-splits over variable values, allowing simpler constraints to be targeted to sub-sets of solution variables.

The translation of first order axioms and the constraints implied by the machine learning system was very successful in reformulating CSPs into more efficient models (for details of the speed-ups obtained, see [2]). In addition, the translation suite was extended to deal with more complex algebras with multiple operators, such as rings, and to consider additional background concepts such as inverses. In the next section, we describe advances made since [2] which have enabled the translation suite to function in non-algebraic domains.

## 3 Non Domain-Specific Translation

To broaden the application of the translation suite to more general problem domains, we have extended its capabilities, as described below. Specifically, the translation suite can now translate more general predicate logic, and we aim for the system to eventually handle full first order logic problem specifications. A standard CSP in the domain of finite algebras involves instantiating the results of the operators on particular pairs of elements, i.e., filling in the Latin square with integers representing the result element. An alternative method would be to consider the operator as an arity 3 predicate, where $*(a, b, n)$ is true when $a * b = n$, and false otherwise. For example, if $2 * 3 = 1$ then $*(2, 3, 1)$ is true and any other combination $*(2, 3, i)$ for $i \neq 1$ would be false. We apply a similar idea to the translation of first-order predicate logic sentences. To describe this approach, we use the well known "who killed Aunt Agatha" problem as a running example. A first order formulation of this problem is as follows (where *killed(x,y)*, *hates(x,y)* and *is_richer_than(x,y)* represent *x killed y*, *x hates y* and *x is richer than y* respectively, with all other predicate meanings obvious):

$lives\_in\_mansion(agatha)$.
$lives\_in\_mansion(butler)$.
$lives\_in\_mansion(charles)$.
$\forall\, x\ (hates(agatha, x) \rightarrow \neg(hates(charles, x)))$.
$hates(agatha, agatha)$.
$hates(agatha, charles)$.
$\forall\, x\ (hates(agatha, x) \rightarrow hates(butler, x))$.
$\forall\, x\ y\ (killed(x, y) \rightarrow hates(x, y))$.
$\forall\, x\ y\ (killed(x, y) \rightarrow \neg(is\_richer\_than(x, y)))$.
$\forall\, x\ (\neg(is\_richer\_than(x, agatha)) \rightarrow hates(butler, x))$.
$\forall\, x\ (\neg(hates(agatha, x)) \leftrightarrow x = butler)$.
$\exists\, x\ (lives\_in\_mansion(x) \wedge killed(x, agatha))$.

Note that the problem is to determine who – of the three suspects living in the mansion – killed Aunt Agatha. The translation routine determines both the model (variables and domain) for the CSP expression of the problem and the constraints that should be generated. Previously, the variable and domain information was provided by the user according to the properties of algebra being studied. In order to model the problem as a CSP, the suite first identifies the predicates that have been used within the list of sentences. These will be of the form $pred\_name(variable\_list)$ and are highlighted during the first part of the translation process, when the sentences are parsed using a Definite Clause Grammar. In the partially-translated, nested expression, the syntax $pred(predicate\_name, [parameter\_list])$ explicitly identifies predicates. Next, the translator identifies all constants within the problem. For the Aunt Agatha problem, the system identifies four predicates, namely hates/2, killed/2, lives_in_mansion/2 and is_richer_than/2, and it identifies three constants, namely: agatha, butler and charles.

If we apply a closed-world assumption, there is enough information to model the problem variables. In the CSP model, one boolean variable is declared for each possible combination of constants for each predicate. For example, three boolean variables are created for the $lives\_in\_mansion$ predicate, namely one for each of $agatha$, $butler$

and $charles$. Similarly, nine boolean variables are declared for each possible combination of the $killed$ predicate, which has arity two, and so on, with thirty variables declared in total. We believe this could be reduced by further careful analysis of the interaction of predicates and this will be considered in future work.

Once the model has been established, the individual sentences are converted into the CLPFD predicates which post constraints on the variables within that model. The method is similar to that described in [2], for instance, universal quantification is translated by introducing predicates to consider each particular combination of possible values. However, in these more general translations, the domain of the CSP variables is not determined by the size of, for example, the particular algebra but, rather, by the number of constants declared in the problem. In addition, we are no longer trying to determine integer values for the problem variables. Instead, the solver must determine the truth value, as a 0 or 1 for each combination of each predicate.

Consider the logic sentence $all\ y\ (killed(agatha, y) \rightarrow hates(agatha, y))$. The universal quantifier is translated in the same manner as in §2 by generating a solver predicate to generate a 3 element list, one element of which considers each of the cases for the value of y: $agatha$, $butler$ and $charles$. When this predicate is called by the solver, the following three constraints will be posted:

$killed(agatha, agatha) \rightarrow hates(agatha, agatha).$
$killed(agatha, butler) \rightarrow hates(agatha, butler).$
$killed(agatha, charles) \rightarrow hates(agatha, charles).$

Given this, the sentence $killed(agatha, y) \rightarrow hates(agatha, y)$ is translated into the three logic programs shown below. When the CSP model is run, $E_1$ would be posted three times with variable V1 taking each of the three constant values.

```
E_1(V1,MT):-
 E_2(V1,MT,RV1),
 E_3(V1,MT,RV2),
 RV1 #=> RV2.

E_2(V1,MT,RV0):-
 pred_var_at(killed,
   [agatha,V1],MT,PV1),
 PV1 #= 1 #<=> RV0.

E_3(V1,MT,RV0):-
 pred_var_at(hates,
   [agatha,V1],MT,PV1),
 PV1 #= 1 #<=> RV0.
```

In the above translation, predicate E_1 enforces the implication expression. This uses two reification variables RV1 and RV2, as required by CLPFD syntax. Reification refers to the act of representing whether a constraint holds or not with a boolean variable. The reified variables are established in calls to E_2 and E_3. For instance, E_2 reifies variable RV0 with the constraint that the variable representing killed(agatha,x) be equal to 1, i.e., true. Similarly, E_3 reifies hates(agatha,x). The pre-defined predicate $pred\_var\_at$ is responsible for identifying which of the thirty solution variables a particular predicate and variable combination represents so that the constraint is applied correctly. This pre-defined predicate makes use of the built-in CLPFD *element* constraint, for similar reasons to those regarding nesting in finite algebras. That is, if one of the parameters is only determined during search, for example existentially quantified variables, then the solution variable it affects can only be resolved at that time, a task for which the *element* constraint is optimized.

When the CSP generated by the translation suite for the Aunt Agatha problem was run in CLPFD, it correctly identified the killer as being Agatha herself. We describe the application of the translation suite to similar investigative problems in §5.

## 4 Application to Comparing AI Systems

The translation suite we developed for use in [2] allows us to take problems framed in Otter syntax and present them to a CSP solving system. There are several other systems that can understand Otter syntax and are capable of solving these problems. The translation suite provides us with a method of assessing the performance of these systems relative to that of our CLPFD system. To demonstrate this, we present the same set of quasigroup existence problems to CLPFD, via our translation suite, and to the model generators MACE2 [6] and MACE4 [7]. Table 1 shows the time taken by each system to complete the search. For each system, we show two results for every algebra. These are the time taken to solve the problem using the system in a basic mode, where we simply state the axioms. The "Full" system result represents the best performance we have managed to achieve by using any additional functionality of that sytem. For example, the CLPFD basic result is a plain translation of the axioms whereas full includes interpreting quasigoups as latin squares and using the automated reformulation technique described in [2], where more details of the translation and solution process are provided.

| Algebra | size | CLPFD Basic | CLPFD Full | MACE2 Basic | MACE2 Full | MACE4 Basic | MACE4 Full |
|---|---|---|---|---|---|---|---|
| QG1 | 4 | 1,050 | 1,050 | 10 | - | 250 | 80 |
| | 5 | 101,220 | 98,490 | 1,020 | 40 | 112,230 | 3,720 |
| QG2 | 4 | 1,000 | 990 | 10 | - | 270 | 80 |
| | 5 | 66,120 | 65,150 | 700 | 20 | 102,370 | 2,750 |
| QG3 | 6 | 6,030 | 1,880 | 520 | - | 680 | - |
| | 7 | 353,480 | 84,480 | 105,410 | 20 | 124,310 | 30 |
| QG4 | 6 | 15,910 | 4,130 | 540 | 10 | 710 | 10 |
| | 7 | 1,466,650 | 318,000 | 129,540 | 180 | 142,840 | 100 |
| QG5 | 6 | 5,810 | 3,130 | 140 | - | 200 | 10 |
| | 7 | 112,270 | 77,790 | 7,760 | 10 | 9,900 | 30 |
| QG6 | 8 | 131,230 | 14,960 | 9,620 | - | 29,570 | - |
| | 9 | 3,397,720 | 385,550 | 659,420 | 10 | 2,618,720 | 20 |
| QG7 | 7 | 73,060 | 5,670 | 620 | 10 | 930 | 10 |
| | 8 | 2,543,250 | 213,390 | 20,330 | 20 | 51,050 | 40 |

**Table 1.** Solver times (ms) for finding all solutions

Quasigroup existence is one of the benchmark problems from the CSPLib [4] library of CSPs and one might expect a CSP solver would be the most appropriate system to use. However, the model generators easily outperform CLPFD for these problems, as the model generators are implemented in C and are highly tailored. It is interesting to note one of the differences between MACE2 and MACE4. The former solves the equivalent propositional problem using the Davis-Putnam-Loveland-Logemann algorithm, whereas MACE4 converts the input into an equivalent set of ground clauses and uses these to apply a recursive backtracking search procedure. This bears many similarities to CSP solvers, for example it considers assignments to individual 'cells' (solution variables), one by one, propagating them to other cells using the ground input clauses. It also provides analogues of search heuristics such as smallest-domain.

## 5 Application to Investigative Reasoning

By the term *investigative reasoning*, we mean the solving of problems where the goal is to determine which of a number of candidates is causing a particular phenomenon. The Aunt Agatha problem described above is such a problem, relating to criminology, where the phenomenon was a murder and the candidates were the three suspects. This type of problem is ubiquitous. Consider, for instance, a patient presenting certain symptoms (the phenomena) and a doctor having to determine which of a number of diseases (the candidates) the patient has. Moreover, when on a regime of various drugs (the candidates), the patient might suffer a side-effect (the phenomenon), and the doctor may once again have to perform an investigation.

Working within the standard AI paradigm, the technique to use would be dictated by the background knowledge available, and once chosen, it is unlikely that any other technique would be considered. In particular, if there are a set of constraints that the guilty candidate must satisfy, then one would naturally apply a constraint solver to a CSP with one variable, the domain of which consists of the candidates. If however, there are axioms and theorems relating the candidates to the phenomenon, then it might be more natural to apply a theorem prover to the problem. To expand upon this, note that the Aunt Agatha problem was taken from the TPTP library [9] of problems for first order *theorem provers*. Stated there, the problem is expressed as a need to prove that Aunt Agatha killed herself. A more difficult problem is, of course, to determine who killed Aunt Agatha. To use a theorem prover to solve this is purely a matter of repeatedly calling the theorem prover to attempt to prove that (a) agatha killed agatha (b) butler killed agatha (c) charles killed agatha. As only one of these attempts results in a proof, the problem is solved. Finally, if the background knowledge includes examples of previous cases when there was a similar phenomenon and a similar set of candiates, then a profile for the guilty candidate from those case-studies could be machine-learned. On applying the profile to the candidates for the current case, if only one candidate satisfied the profile, then the problem would also have been solved.

We are interested in the situation when all three types of background information is available, as this much more closely represents the situation in the real world. For instance, if one imagines a police officer investigating a crime, they will be able to refer to similar cases from their past; they will have constraints on the suspects, such as location; and they will have facts relating candidates to the phenomena, e.g., motives. Therefore, it is highly likely that they will use a combination of deductive, inductive and constraint-based reasoning to solve the case. To employ AI techniques to simulate such a combined approach to investigative reasoning, it is imperative to translate problems between the syntaxes of the various AI systems employed. To begin our work towards implementing a combined reasoning system for investigative reasoning, we used the extended translation suite described in §3 to restate automated theorem proving problems as CSPs and to solve those theorems using a CSP solver.

The TPTP library of problems for theorem provers [9] contains a large number of problems for testing and benchmarking theorem provers. We have used our system to attempt to prove the problems in the "Puzzles" category, which contains 61 problems. The SystemOnTPTP website (www.cs.miami.edu/ tptp/cgi-bin/SystemOnTPTPFormMaker) allows users to convert each problem into Otter syntax, which we used as a starting point

for the input into our translation suite. This input requires some reformatting before it can be used by the suite. For example, the suite currently requires explicit universal quantification and variables must be renamed to be one of a subset of allowable variable names. Note also that Otter attempts to prove theorems by finding an inconsistency between the background information and the negated theorem, which is known as a refutation proof. The user of the translation suite has various options for how they present the theorem. They can include the negated theorem and, if the system fails to find a model they know that the theorem is not consistent with the background information. Alternatively, they may wish to exclude the theorem altogether and just review the values that the solver assigns to various predicates. This is useful in solving puzzles similar to the Aunta Agatha problem of §3, where the user is purely interested in the possible values of the $killed/2$ predicate.

The translation suite was able to prove or solve 18 of the 61 problems in the Puzzles category of the TPTP library. In some instances, we converted purely propositional problems into predicate logic by wrapping the propositional statements in a notional $a\_truth(\ldots)$ predicate. Of the problems the translation suite could not prove, 15 were due to the large number of variables in the problem. The suite currently only accepts a small set of variable names, which should be easy to extend. The suite could not prove 24 of the problems due to nesting of predicates and functions. At present, the suite can only accept predicates in the form $pred\_name(variable\_list)$. These 24 problems included functions as the parameters to predicates and the suite is at present unable to distinguish between the two based upon the context in which they appear. For example, in the predicate $friday(yesterday(X))$, the predicate *friday* gives the truth value of whether or not a given day is *friday* whereas the function $yesterday$ maps one day to another, the preceeding day of the week. The remaining 4 theorems could not be proved for other reasons, such as memory restrictions. We aim to address this by refining the way in which solver models are generated, for example by considering typing of constants. For instance, in a problem relating to people and days of the week we would not be interested in the truth value of, say, $has\_beard(friday)$.

Note that, to further continue our study of investigative reasoning, we have hand-translated the Aunt Agatha problem into a machine learning problem and used the Progol ILP system [8] to correctly learn a profile (from synthetic case studies) which was true only of Aunt Agatha, thus solving the problem. Moreover, we are currently engaged in a project which aims to build a system able to automatically generate such whodunnit problems so that Otter, CLPFD and Progol can solve the problem. The next step will be to combine AI systems in a fruitful way to solve the problems more efficiently than using single systems. We believe this will be particularly useful in situations when only partial background knowledge is available (which again more closely simulates real-world problems). In such situations, while individual systems might not have enough constraints, or theorems, or case studies to solve the problem, their combined efforts in narrowing down the candidates might lead to just one candidate being left. We also plan more sophisticated combinations of the systems. For instance, we plan to investigate how a machine learning system could be used to learn some missing constraints or theorems empirically from case-studies. This additional information might enable one of the systems to solve the problem even though it started with insufficient information.

## 6 Conclusions and Future Work

We have described a translation suite able to take problem specifications expressed in first order logic and turn them into CSPs for the Sicstus CLPFD constraint solver. As discussed in [2], while Sicstus is a first order system, translating problems as efficient CSPs is not straightforward. We describe recent extensions to the suite which have enabled it to translate a larger class of problems. We demonstrated the utility of this with two applications. Firstly, by comparing model generators and constraint solvers, we showed that meaningful comparisons can be made between AI systems, which, if nothing else, provides more information to potential consumers of AI techniques. Secondly, by applying constraint solving to theorem proving problems, we hinted at the possibility of combining AI systems so that the whole is more than the sum of the parts. We argued that, for problems with various types of background information, in particular investigation problems where the goal is to identify a guilty suspect, combining reasoning systems has much potential, especially if the background information is incomplete.

We plan to extend the range of translatable problems. In particular, we aim for the system to take any first order problem specification and translate it to the CLPFD syntax, addressing the issues raised in the investigative reasoning application described above. Moreover, while it has been advantageous to work with first order systems, we plan to enable the systen to translate second and higher order logic problem specifications into the CLPFD syntax, and into the syntax of other more powerful constraint solvers. We also plan to reverse the translation, e.g., from CLPFD to Otter syntax, which will increase fluidity in our AI applications. We have begun to build a system able to generate synthetic investigation problems able to be solved by CLPFD, Otter and Progol. On completion of this system, we aim to use the problems generated to test the hypothesis that combined reasoning systems are more efficient at solving AI problems than stand-alone systems, and combined reasoning is the only option when the background information is incomplete.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. We wish to thank Dr. Ian Miguel for his help and many contributions to this work.

## References

1. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.
2. J. Charnley, S. Colton, and I. Miguel. Automatic generation of implied constraints. In *Proceedings of the 17th European Conference on Artificial Intelligence*, 2006.
3. S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.
4. I. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, APES-09-1999, 1999.
5. W. McCune. The Otter user's guide. Technical Report ANL/90/9, ANL, 1990.
6. W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical report, ANL/MCS-TM-194, 1994.
7. W. McCune. MACE4 Manual and Guide. Technical Report ANL/MCS-TM-264, ANL, 2003.
8. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
9. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.