Constraint Modelling: A Challenge for First Order Automated Reasoning (extended abstract)

John Slaney

NICTA and the Australian National University

Abstract. The process of modelling a problem in a form suitable for solution by constraint satisfaction or operations research techniques, as opposed to the process of solving it once formulated, requires a significant amount of reasoning. Contemporary modelling languages separate the first order description or "model" from its grounding instantiation or "data". Properties of the model independent of the data may thus be established by first order reasoning. In this talk, I survey the opportunities arising from this new application direction for automated deduction, and note some of the formidable obstacles in the way of a practically useful implementation.

1 Constraint Programming

A constraint satisfaction problem (CSP) is normally described in the following terms: given a finite set of decision variables $v_1, \ldots, v_n$ with associated domains $D_1, \ldots, D_n$, and a relation $C(v_1, \ldots v_n)$ betwen the variables, a *state* is an assignment to each variable v_i of a value d_i from D_i. A state is a *solution* to the CSP iff $C(d_1, \ldots, d_i)$ holds. In practice, C is the conjunction of a number of constraints each of which relates a small number of variables. It is common to seek not just any solution, but an optimal one in the sense that it minimises the value of a specified *objective function*.

Mathematical programming is the special case in which the domains are numerical (integers or real numbers) and the constraints are equalities or inequalities between functions (usually polynomial, nearly always linear, in fact) of these. The techniques usable for numbers are so different from those for the general case, however, that CP and MP are often seen as contrasting or even opposing approaches.

Logically, C is a theory in a language in which the v_i are proper names ("constants" in the usual terminology of logic). A state is an interpretation of the language over a domain (or several domains, if the language is many-sorted) corresponding to the domains of the variables, and a solution is an interpretation that satisfies C. On this view, CSP reasoning is the dual of theorem proving: it is seeking to establish possibility (satisfiability) rather than necessity (unsatisfiability of the negation).

Techniques used to solve CSPs range from the purely logical, such as SAT solving, through finite domain (FD) reasoning which similarly consists of a backtracking search over assignments, using a range of propagators appropriate to different constraints to force some notion of local consistency after each assignment, to mixed integer programming using a variety of numerical optimisation algorithms. Hybrid solution methods, in which different solvers are applied to sub-problems, include SMT (satisfiability modulo theories), column generation, large neighbourhood search and many more or less *ad hoc* solver combinations for specific purposes. The whole area has been researched intensively over the last half century, generating an extensive literature from the automated reasoning, artificial intelligence and operations research communities. The reader is referred to Dechter's overview [3] for an introduction to the field.

Constraint programming is an approach to designing software for CSPs, whereby a library of solvers is used in the same maner as libraries of mathematical function computations. The search is controlled by a program written in some high-level language (sometimes a logic programming language, but in modern systems often C++ or something similar) and specific solvers may be used to evaluate particular predicates or perform propagation steps, or may be passed the entire problem after some preprocessing. CP platforms vary in the degree to which they automate control of the propagation queue and the like, or leave it in the hands of the programmer. The constraint programming paradigm gives a great deal of flexibility, allowing techniques to be tailored to problems, while at the same time accessing the power and efficiency of high-performance CSP solvers.

1.1 Separating Modelling from Solving

Engineering a constraint program for a given problem is traditionally a two-phase process. First the problem must be *modelled.* This is a matter of determining what are the decision variables, what are their domains of possible values and what constraints they must satisfy. Then a program must be written to *evaluate* the model by using some solver or combinaton of solvers to search for solutions. Most of the CP and OR literature concerns the second phase, assuming that "the problem" resulting from the modelling phase is given.

In recent years, there has been a growing realisation of the importance of modelling as part of the overall process, so modern CP or MP platforms feature a carefully designed modelling language such as ILOG's OPL [7] or AMPL from Bell Labs [5]. Contemporary work on modelling languages such as ESRA [4], ESSENCE [6] and Zinc [8] aims to provide a rich representation tool, with primitives for manipulating sets, arrays, records and suchlike data structures and with the full expressive power of (at least) first order quantification. It also aims to make the problem representation independent of the solver(s) so that one and the same conceptual model can be mapped to a form suitable for solution by mixed integer programming, by SAT solving or by local search.

In the present report, the modelling language used will be Zinc, which is part of the G12 platform currently under development by NICTA (Australia).

My involvement is in designing and implementing the user environment for G12, which will incorporate theorem proving technology along with much else. The current theorem proving research is joint work with Peter Baumgartner.[1]

1.2 The G12 Platform

The G12 constraint programming platform provides a series of languages and associated tools. At the base is Mercury [10, 2] , a logic programming language with adaptations for contraint logic programming with a propagation queue architecture (`http://www.cs.mu.oz.au/research/mercury/`). Below that are the solvers, which include commercial ones like CPLEX, third party open source ones like MiniSAT and many of our own. The API for incorporating solvers is quite straihtforward. On the top level is the modelling language Zinc, of which more below. Between Mercury and Zinc (remember your periodic table) is Cadmium, a very declarative programming language based on term rewriting, which is designed for mapping one syntax to another and is used in G12 mainly to convert Zinc specifications into simpler ones. For instance, they may be flattened by unrolling quantifiers, or clausified, or expressed as linear programs or as SAT problems.

G12 clearly separates the conceptual model, written in Zinc, from the design model intended to be realised as a Mercury program. It also draws a distinction between the *model*, which is a first order description of the generic problem, and the *data* which are the facts serving to ground the model in a particular instance. The commonly used distinction between facts and rules or "integrity constraints" in deductive databases is somewhat similar.

G12 programs can be related to Zinc models in a spectrum of ways. It is possible to write a constraint program to solve the problem, treating the Zinc specification just as a guide, as is often done in conventional CP. The program can throw the entire problem onto one solver such as CPLEX, or onto MiniSAT as Paradox does, or can mix solvers in arbitrarily fine-grained ways to produce hybrids tailored to individual problems. Alternatively, the program can take the Zinc specification and data as input, in which case we think of it as a way of evaluating the model over the data. It is even possible to avoid writing *any* program, leaving the G12 system itself with its default mappings to do the evaluation and writing only in Zinc. This last represents the closest approach yet to the Holy Grail of high-level programming: one does not program at all; one tells the computer what the problem is, and it replies with the solution.

[1] We have benefitted greatly from being in a team that has included Michael Norrish, Rajeev Gore, Jeremy Dawson, Jia Meng, Anbulagan and Jinbo Huang, and from the presence in the same laboratory of an AI team including Phil Kilby, Jussi Rintanen, Sylvie Thiébaux and others. The G12 project involves well over 20 researchers, including Peter Stuckey, Kim Marriott, Mark Wallace, Toby Walsh, Michael Maher, Andrew Verden and Abdul Sattar. The details of our indebtedness to these people and their colleagues are too intricate to be spelt out here.

Zinc Zinc is a typed (mostly) first order language. It has as basic types `int`, `float` and `bool`, and user-defined finite enumerated types. To these are applied the `set-of`, `array-of`, `tuple`, `record` and subrange type constructors. These may be nested, with some restrictions mainly to avoid such things as infinite arrays and explicitly higher order types (functions with functional arguments). The type `string` is present, but is only used for formatting output. It also allows a certain amount of functional programming, which is not of present interest. It provides facilities for declaring decision variables of most types and constants (parameters) of all types. Standard mathematical functions such as `+` and `sqrt` are built in. Constraints may be written using the expected comparators such as $=$ and $\leq$ or user-defined predicates to form atoms, and the usual boolean connectives and quantifiers (over finite domains) to build up compounds. Assignments are special constraints whereby parameters are given their values. The values of decision variables are not normally fixed in the Zinc specification, but have to be found by some sort of search.

For details, see `http://users.rsise.anu.edu.au/~jks/zinc-spec.pdf`.

Analysing models It is normal to place the Zinc model in one file, and the data (parameters, assignments and perhaps some enumerations) in another. The model tends to stay the same as the data vary. For example, without changing any definitions or general specifications, a new schedule can be designed for each day as fresh information about orders, jobs, customers and prices becomes available.

The user support tools provided by the G12 development environment should facilitate debugging and other reasoning about models independently of any data. However, since the solvers cannot evaluate a model until at least the domains are specified, it is unclear how this can be done. Some static visualisation of the problem, such as views of the Zinc-level constraint graph, can help a little, but to go much further we need a different sort of reasoning.

2 Deductive Tasks

There is no good reason to expect a theorem prover to be used as one of the solvers for the purposes of a constraint programming platform such as G12. Apart from the fact that typical constraint satisfaction problems are trivially satisfiable—the main issue is optimality, not the existence of solutions—the reasoning required amounts to propagation of constraints over finite domains rather than to chaining together complex inferences. For this purpose SAT solvers and the like are useful, but traditional first order provers are not. However, for analysing the models before they have been grounded by data, first order deduction is the only option. The following tasks are all capable of automation:

1. Proof that the model is inconsistent.
   Inconsistency can indicate a bug, or merely a problem overconstrained by too many requirements. It can arise in "what if" reasoning, where the programmer has added speculative conditions to the basic description or it can

arise where partial problem descriptions from different sources have been combined without ensuring that their background assumptions mesh. Often, inconsistency does not afflict the pure model, but arises only after part of the data has been added, so detecting it can require a certain amount of grounded reasoning as well as first order unification-driven inference.

   A traditional debugging move, also useful in the other cases of inconsistency, is to find and present a [near] minimal inconsistent core: that is, a minimally inconsistent subset of the constraints. The problem of "axiom pinpointing" in reasoning about large databases is similar, except that in the constraint programming case the number of possible axioms tends to be comparatively small and the proofs of inconsistency comparatively long. The advantage of finding a first order proof of inconsistency, rather than merely analysing nogoods from a backtracking search, is that there is some hope of presenting the proof to a programmer, thus answering the question of *why* the particular subset of constraints is inconsistent.

2. Proof of symmetry.

   The detection and removal of symmetries is of enormous importance to finite domain search. Where there exist isomorphic solutions, there exist also isomorphic subtrees of the search tree. Moreover, there can be thousands or millions of solutions isomorphic to a given one, meaning that almost all of the search is a waste of time and can be eliminated if the symmetries are detected early enough. A standard technique is to introduce "symmetry breakers", which are extra constraints imposing conditions satisfied by some but not all (preferably exactly one) of the solutions in a symmetry class. Symmetry breakers prevent entry to subtrees of the search tree isomorphic to the canonical one.

   It may be evident to the constraint programmer that some transformation gives rise to a symmetry. Rotating or reflecting the board in the N Queens problem would be an example. However, other cases may be less obvious, especially where there are side constraints that could interfere with symmetry. Moreover, it may be unclear whether the intuitively obvious symmetry has been properly encoded or whether in fact every possible solution can be transformed into one which satisfies all of the imposed symmetry breakers.

   It is therefore important to be able to show that a given transformation defined over the state space of the problem does actually preserve the constraints, and therefore that it transforms solutions into solutions. Since symmetry breakers may be part of the model rather than part of the data, we may wish to prove such a property independently of details such as domain sizes. There is an example in the next section.

3. Redundancy tests.

   A redundant constraint is one that is a logical consequence of the rest. It is common to add redundant constraints to a problem specification, usually in order to increase the effect of propagation at each node of the search tree. Sometimes, however, redundancy may be unintentional: this may indicate a bug—perhaps an intended symmetry-breaker which in fact changes nothing—or just a clumsy encoding. Some constraints which are not redundant in the model may, of course, become redundant when the data are added.

Where redundant constraints are detected, either during analysis of the model or during preprocessing of the problem including data, this might usefully be reported to the constraint programmer who can then decide whether such redundancy is intentional, whether it matters or not, and whether the model should be adjusted in the light of this information. It may also be useful to report irredundancy where a supposedly redundant constraint has been added: the programmer might usefully be able to request a redundancy proof in such a case.

4. Functional dependency.

   The analogue at the level of functions of redundancy at the level of propositions is dependency in the sense that the values of certain functions may completely determine the value of another for all possible arguments. As in the case of constraint redundancy, functional dependence may be intentional or accidental, and either way it ay be useful to the constraint programmer to know whether a function is dependent or not.

   Consider graph colouring as an example. It is obvious that in general (that is, independently of the graph in question) the extensions of all but one of the colours are sufficient to fix the extension of the final one, but that this is not true of any proper subset of the "all but one". In the presence of side constraints, however, and especially of symmetry breakers, this may not be obvious at all. In such cases, theorem proving is the appropriate technology.

5. Equivalence of models.

   It is very common in constraint programming that different approaches to a given problem may result in *very* different encodings, expressing constraints in different forms and even using different signatures and different types. The problem of deciding whether two models are equivalent, even in the weak sense that solutions exist for the same values of some parameters such as domain sizes, is in general hard. Indeed, in the worst case, it is undecidable. However, hardness in that sense is nothing new for theorem proving, so there is reason to hope that equivalence can often enough be established by the means commonly used in automated reasoning about axiomatisations.

6. Simplification

   A special case of redundancy, which in turn is a special case of model equivalence, occurs in circumstances where the full strength of a constraint is not required. A common example is that of a biconditional ($\Leftrightarrow$) where in fact one half of it ($\Rightarrow$) would be sufficient. Naïve translation between problem formulations can easily lead to unnecessarily complicated constraints such as $x < \sup(S)$ which is naturally rendered as $\exists y \in S((\forall z \in S.z \leq y) \wedge x < y)$, while the simpler $\exists y \in S.x < y$ would do just as well. Formal proofs of the correctness of simplifications can usefully be offered to the programmer at the model analysis stage.

```
int: N;
array[1..N] of var int: q;
constraint forall (x in 1..N, y in 1..x-1) (q[x] != q[y]);
constraint forall (x in 1..N, y in 1..x-1) (q[x]+x != q[y]+y);
constraint forall (x in 1..N, y in 1..x-1) (q[x]-x != q[y]-y);
solve satisfy;
```

Fig. 1. Zinc model for the N Queens problem

2.1 Preliminary experiments

Our experiments are very much work in progress, so we are in a position only to report preliminary findings rather than the final word. Here are comments on just two toy examples, more to give a flavour than to present systematic results.

Proving symmetry We consider the N Queens problem, a staple of CSP reasoning. N queens are to be placed on an a chessboard of size $N \times N$ in such a way that no queen attacks any other along any row, column or diagonal. The model is given in Figure 1 and the data consists of one line giving the value of N (e.g. '`N = 8;`'). Suppose that as a result of inspection of this problem for small values of N it is conjectured[2] that the transformation $s[x] = q[n+1-x]$ is a symmetry. We wish to prove this for all values of N. That is, we need a first order proof that the constraints with s substituted for q follow from the model as given and the definition of s. Intuitively, the result is obvious, as it corresponds to the operation of reflecting the board, but intuitive obviousness is not proof and we wish to see what a standard theorem prover makes of it.

The prover we took off the shelf for this experiment was Prover9 by McCune [9]. Clearly a certain amount of numerical reasoning is required, for which additional axioms must be supplied. The full theory of the integers is not needed: algebraic properties of addition and subtraction, along with numerical order, seem to be sufficient. All of this is captured in the theory of totally ordered abelian groups, which is quite convenient for first order reasoning in the style of Prover9. We tried two encodings: one in terms of the order relation $\leq$ and the other an equational version in terms of the lattice operations `max` and `min`.

The first three goals:
$(1 \leq x \wedge x \leq n) \Rightarrow 1 \leq s(x)$
$(1 \leq x \wedge x \leq n) \Rightarrow s(x) \leq n$
$s(x) = s(y) \Rightarrow x = y$
are quite easy for Prover9 when $s(x)$ is defined as $q(n+1-x)$. By contrast, the other two
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) + x \neq s(y) + y$
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) - x \neq s(y) - y$
are not provable inside a time limit of 30 minutes, even with numerous helpful lemmas and weight specifications to deflect useless subformulae like $q(q(x))$ and

[2] In fact, there is experimental software to come up with such conjectures automatically.

$q(n)$. It makes little difference to these results whether the abelian l-group axioms are presented in terms of the order relation or as equations.

To push the investigation one more step, we also considered the transformation obtained by setting s to q^{-1}. This is also a symmetry, corresponding to reflection of the board about a diagonal, or rotation through 90^o followed by reflection as above. This time, it was necessary to add an axiom to the Queens problem definition, as the all-different constraint on q is not inherited by s. The reason is that for all we can say in the first order vocabulary, N might be infinite—e.g. it could be any infinite number in a nonstandard model of the integers—and in that case a function from $\{1 \dots N\}$ to $\{1 \dots N\}$ could be injective without being surjective.

The immediate fix is to add surjectivity of the 'q' function to the problem definition, after which in the relational formulation Prover9 can easily deduce the three small goals and the first of the two diagonal conditions. The second is beyond it, until we add the redundant axiom

$x_1 - y_1 = x_2 - y_2 \Rightarrow x_1 - x_2 = y_1 - y_2$

With this, it finds a proof in a second or so. In the equational formulation, no proofs are found in reasonable time.

The more general issue, however, is that many CSP encodings make implicit use of the fact that domains are finite, as a result of which it may be impossible to deduce important properties by first-order reasoning without fixing bounds on parameters. If theorem proving is to be a useful tool in G12, ways will have to be found to circumvent such difficulties, issuing warnings if necessary.

Another message from the experiments is that a lot of arithmetical reasoning tricks and transformations will have to be identified and coded into the system. The above transformation of equalities between differences (and its counterparts for inequalities) illustrates this.

An encouraging feature is that a considerable amount of the reasoning turns only on algebraic properties of the number systems, and so may be amenable to treatment by standard first order provers.

Proving redundancy A toy example of redundant constraints is found in the following logic puzzle [1]:

> On June 1st, five couples will celebrate their wedding anniversaries. Their surnames are Johnstone, Parker, Watson, Graves and Shearer. The husbands' given names are Russell, Douglas, Charles, Peter and Everett. The wives' given names are Elaine, Joyce, Marcia, Elizabeth and Mildred.
>
> 1. Joyce has not been married as long as Charles or the Parkers, but longer than Douglas and the Johnstones.
> 2. Elizabeth has been married twice as long as the Watsons, but only half as long as Russell.
> 3. The Shearers have been married ten years longer than Peter and ten years less than Marcia.
> 4. Douglas and Mildred have been married for 25 years less than the Graves who, having been married for 30 years, are the couple who have been married the longest.

5. Neither Elaine nor the Johnstones have been married the shortest amount of time.
6. Everett has been married for 25 years

Who is married to whom, and how long have they been married?

Parts of clue 1, that Joyce has been married longer than Douglas and also longer than the Johnstones, are deducible from the other clues. Half of clue 5, that Elaine has not been married the shortest amount of time, is also redundant. The argument is not very difficult: a little arithmetical reasoning establishes that the five numbers of years married are 5, 10, 20, 25 and 30 (three of them are 30, 5 and 25, and the five contain a sequence of the form $x, 2x, 4x$). Mildred has been married for 5 years (clue 4) from which it quickly follows that both Elaine and Joyce have been married for longer than Mildred and therefore than Douglas. That Joyce has been married for longer than the Johnstones is a slightly more obscure consequence of the other clues, but a finite domain constraint solver has no difficulty with it.

Presenting the problem of deriving any of these redundancies to Prover9 is not easy. The small amount of arithmetic involved is enough to require a painful amount of axiomatisation, and even when the addition table for the natural numbers up to 30 is completely spelt out, making use of that to derive simple facts such as those above is beyond the abilities of the prover. Even given an extra clause stating that the five numbers involved are 5, 10, 20, 25 and 30, in ten thousand iterations of the given clause loop it gets nowhere near deducing that Joyce has been married for longer than Douglas.

If the fact that the numbers of years are all in the set $\{5, 10, 15, 20, 25, 30\}$ is given as an axiom, *and* extra arguments are given to all function and relation symbols to prevent unification across sorts, then of course the redundancy proofs become easy for the prover. However, it is unreasonable to expect that so much help will be forthcoming in general. Even requiring just a little of the numerical reasoning to be carried out by the prover takes the problem out of range.

Part of the difficulty is due to the lack of numerical reasoning, but as before, forcing the problem statement into a single-sorted logic causes dramatic inefficiency. It is also worth noting that the proofs of redundancy are long (some hundreds of lines) and involve nearly all of the assumptions, indicating that axiom pinpointing is likely to be useless for explaining overconstrainedness at least in some range of cases.

2.2 Conclusions

While, as noted, the investigation is still preliminary, some conclusions can already be drawn. Notably, work is required on expanding the capacities of conventional automatic theorem provers:

1. Numerical reasoning, both discrete and continuous, is essential. The theorems involved are not deep—showing that a simple transformation like reversing the order $1 \ldots N$ is a homomorphism on a model or restricting

attention to numbers divisible by 5—but are not easy for standard theorem proving technology either. Theorem provers will not succeed in analysing constraint models until this hurdle is cleared.

2. Other features of the rich representation language also call for specialised reasoning. Notably, the vocabulary of set theory is pervasive in CSP models, but normal theorem provers have difficulties with the most elementary of set properties. Some first order reasoning technology akin to SMT, whereby specialist modules return information about sets, arrays, tuples, numbers, etc. which a resolution-based theorem prover can use, is strongly indicated. Theory resolution is the obvious starting point, but is it enough?
3. Many-sorted logic is absolutely required. There are theorem provers able to exploit sorts, but most do not—a telling point is that TPTP still does not incorporate sorts in its notation or its problems.
4. Constraint models sometimes depend on the finiteness of parameters. Simple facts about them may be unprovable without additional constraints to capture the effects of this, as illustrated by the case of the symmetries of the N Queens problem. This is not a challenge for theorem provers as such but rather for the process of preparing constraint models for first order reasoning.
5. In some cases, proofs need to be presented to human programmers who are not working in the vocabulary of theorem proving, who are not logicians, and who are not interested in working out the details of complicated pramodulation inferences. Despite some efforts, the state of the art in proof presentation remains lamentable. This *must* be addressed somehow.

Despite the above challenges, and perhaps in a sense because of them, constraint model analysis offers an exciting range of potential rôles for automated deduction. Constrait-based reasoning has far wider application than most canvassed uses of theorem provers, such as software verification, and certainly connects with practical concerns much more readily than most of [automated] pure mathematics. Reasoning about constraint models without their data is a niche that only first (or higher) order deductive systems can fill. Those of us who are concerned to find practical applications for automated reasoning should be working to help them fill it.

References

1. Anonymous. Anniversaries: Logic puzzle. `http://www.genealogyworldwide.com/genealogy_fun.php`.
2. Ralph Becket, Maria Garcia de la Banda, Kim Marriott, Zoltan Somogyi, Peter Stuckey, and Mark Wallace. Adding constraint solving to Mercury. In *Proceedings of the Eighth International Symposium on Practical Aspects of Declarative languages*. Springer Verlag, 2006.
3. Rina Dechter and David Cohen. *Constraint Processing*. Morgan Kaufmann, 2003.
4. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR'03, Revised Selected Papers (LNCS 3018)*, pages 214–232. Springer-Verlag, 2004.

5. Robert Fourer, David Gay, , and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002. `http://www.ampl.com/`.
6. A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 80–87, 2007.
7. Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, 1999.
8. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints, Special Issue on Abstraction and Automation in Constraint Modelling*, 13(3), 2008.
9. William McCune. Prover9 and mace4. `http://www.cs.unm.edu/ mccune/mace4/`.
10. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, 1995. `http://www.cs.mu.oz.au/research/mercury/information/papers.html`.