Constraint Modelling: A Challenge for Automated Reasoning

Peter Baumgartner and John Slaney
{*firstname.secondname*}@nicta.com.au

NICTA[⋆] and Australian National University, Canberra, Australia

Abstract. Cadoli *et al* [BCM04,MC05,CM04] noted the potential of first order automated reasoning for the purpose of analysing constraint models, and reported some encouraging initial experimental results. We are currently pursuing a very similar research program with a view to incorporating deductive technology in a state of the art constraint programming platform. Here we outline our own view of this application direction and discuss new empirical findings on a more extensive range of problems than those considered in the previous literature. While the opportunities presented by reasoning about constraint models are indeed exciting, we also find that there are formidable obstacles in the way of a practicaly 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*.

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

⋆ NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative.

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 [DC03,MS98] for an introduction to the field.

Constraint programming is an approach to designing software for CSPs, whereby 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. 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 produced to *evaluate* the model by using some solver or combinaton of solvers to search for solutions. This program may be written by a human programmer, or derived automatically from the model, or some combination of the two. Most of the Constraint Programming (CP) and Operations Research (OR) literature concerns problem solving, 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 Mathematical Programming (MP) platforms feature a carefully designed modelling language such as ILOG's OPL [Hen99] or AMPL from Bell Labs [FGK02]. Contemporary work on modelling languages such as ESRA [FPg04], ESSENCE [FGJ$^+$07] and Zinc [MNR$^+$08] 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.

1.2 Zinc

In the present report, the modelling language used will be Zinc, which is part of the G12 platform currently under development by NICTA (Australia).[1]

The G12 platform provides a series of languages: Mercury, Cadmium and Zinc. Mercury is a constraint logic programming language, Cadmium a rather specialised programming language for syntax transormations based on term rewriting, and Zinc a modelling language in which problems are specified in an algorithm-independent way [MNR+08]. It is a typed (mostly) first order language, with 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). Zinc 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.

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: we need first order deduction.

[1] See `http://nicta.com.au/research/projects/constraint_programming_platform`. 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.

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. In many practical cases the main issue is optimality, the existence of solutions being obvious, and it is not clear how theorem proving can help with this. Moreover, the reasoning required to solve CSPs typically amounts to propagation of constraints over finite domains rather than to chaining together complex inferences, and for this purpose SAT solvers and the like are useful, but traditional first order provers are not.[2] However, for analysing the models before they have been grounded by data, first order deduction is the only option. Previous work [BCM04,MC05,CM04] has identified some tasks and practical experiences using a first-order theorem prover. A serious deficiency of the previous accounts, however, is the absence of numerical reasoning. Zinc, like other modelling languages, supports integer domains, and even floating point ones. These are crucial: there is no hope of dealing adequately with industrial problems of scheduling and resource management without numbers. However, as we show below, even very simple integer arithmetic poses major difficulties for first-order theorem provers.

We are interested in the following problems, which are all capable of automation.

2.1 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.

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 logically similar, but 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 a proof can be presented to a programmer, thus answering the question of *why* the particular subset of constraints is inconsistent.

[2] The "typical" case is not the only case, of course. The satisfiability problem for Zinc is undecidable, since the language can express Diophantine equations over the unbounded domain of the integers. For Zinc models (without data) it is even easier to find undecidable theories, since the problem of deciding whether an arbitrary first order formula has a finite model is easily encoded, as are special cases like the word problem for semigroups. Sometimes, therefore, theorem proving may be the best we can do, but such cases do not arise in industrial process scheduling or other common CP applications.

2.2 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. In some cases almost all of the search 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 by 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.

2.3 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.

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 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.

2.4 Functional Dependency

Functions may also be redundant, 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 may 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.

2.5 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.

Concrete applications of proving equivalence stem from all sorts of transformations of constraint models. For instance, one might (automatically) detect that certain variables must receive different values according to the current model and pose a global `all_different` constraint instead. Other transformations are inspired by optimising compiler technology, such as loop-invariants hoisting (exchange "forall" and "exists" loops), common subexpression elimination, algebraic rewriting (theory specific equational rewriting) and partial evaluation (see [MKB$^+$05]).

2.6 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 $a < \sup(S)$ which is naturally rendered as
$\exists y(\forall z((\forall x \in S(x \leq z)) \leftrightarrow y \leq z) \wedge a < y)$,
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 1..N: q;
constraint forall (x in 1..N, y in 1..x-1)
    (q[x] != q[y]
∧   (q[x]+x != q[y]+y
∧   (q[x]-x != q[y]-y);
solve satisfy;
```

Fig. 1. Zinc model for the N Queens problem

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

Fig. 2. Alternative model for the N Queens problem

3 Experiments

We conducted some experiments in order to evaluate the feasibility of state of the art automated reasoning technology to solve deductive proof tasks as explained in Section 2.

3.1 N-Queens

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;`').

Index Refinement As a very simple example of equivalence of models, consider the formulation of the n-queens problem in Figure 2. Notice how it differs slightly from the one in Figure 1 in the use of indexing. One may expect that re-formulations like these occur frequently and their correctness should be rather straightforward to establish automatically.

Alldifferent Constraint. The `alldifferent` constraint on a set of variables requires them to take pairwise different values. Because specialized, efficient constraint solving techniques have been developed for `alldifferent`, it may make sense to replace or enrich parts of a given constraint model by an `alldifferent` constraint. Clearly, in our example, any solution of the n-queens problem obviously satisfies the `alldifferent` constraint for $\{\mathtt{q}[1], \ldots, \mathtt{q}[N]\}$. It is easy to formulate this as a proof task: simply add the constraint `not(forall (x in 1..N, y in 1..x-1) (q[x] != q[y]))` to the constraint model and prove unsatisfiability. Of course, a sufficiently rich set of axioms for the underlying theories (integer arithmetic, e.g.) has to be provided to the prover as well.

Detecting Symmetries Suppose that as a result of inspection of this problem for small values of N it is conjectured, either automatically or by the programmer, 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, this 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.

One prover we took off the shelf for this experiment was Prover9 by McCune [McC].[3] 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, suffice. All of this is captured in the theory of totally ordered abelian groups (see e.g. [MA88]) which is quite convenient for first order reasoning [Wal01]. 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. 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. This time, it is 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—it could be any infinite number in a nonstandard model of the integers—and in that case a function from $\{1 \ldots N\}$ to $\{1 \ldots N\}$ 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

[3] Previous work [CM04,CM05] user Otter for similar problems in graph coloring; its successor Prover9 is similar but generally superior.

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.

We also tried the Vampire prover (version 8) and came to the same conclusions as with Prover9. With redundant axioms the proof is found easily, without them, not. One message from this experiment is that care must be taken to avoid implicit appeal to the fact that domains are finite. Another is that a range 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.

A perhaps even more natural idea is to try theorem provers with native support for arithmetic reasoning instead of "general" first-order logic theorem provers. The development of such provers is still an active research topic, see, e.g., [BFT08,KV07,WP06,Rüm08], but some of the available SMT-solvers (Satisfiability Modulo Theories [RT06]) already support the logic and theories that we need. To explain, the proof obligation in the example is an entailment between two universally quantified formulas with free functions symbols, over the theory of linear integer arithmetic. SMT solvers are not full-fledged theorem provers for first-order logic, and on proof tasks of that form, current SMT solvers need to rely on (incomplete) instantiation heuristics to remove the universal quantifiers in the premise of an entailment. Despite that, in the example, the two SMT solvers that we tried, CVC3 [BT07] and Yices, had no difficulties even with the *original* problem formulation, the one without any additional redundant axioms (runtimes: less than one second). We found this a very encouraging result.

We also tried the default solver that comes with the G12 platform. Like any constraint solver, it is not a theorem prover and cannot prove that the symmetry property holds for all values of the board size N. However, we found it instructive to prove, with G12, the symmetry property with specific values for N. The rationale behind this exercise is the methodology to first try some small instances, to see if a conjecture is trivially falsified. Of course this is pointless in this example, but in general it may help to find bugs in the coding, or counterexamples for non-valid conjectures.

Table 1 summarizes the experimental results, for all problems described above. The results indicate that the SMT solvers, YICES and CVC3, perform much better on these problems than the theorem provers. Moreover, the formulations for the theorem provers are highly sensitive to the axiomatization of the background theory. Without a minimal "right" set of axioms, the proof will not be found or proof times increase drastically.

Problem	E	E-Darwin	SPASS	Vampire	YICES	CVC3	G12
`alldifferent` implied	-	< 1	1	< 1	< 1	< 1	< 1 ($N = 10$) 5 ($N = 12$) 137 ($N = 14$) > 600 ($N = 16$)
Index refinement	-	-	- ($\Leftrightarrow$) 6 ($\Leftarrow$) 12 ($\Rightarrow$)	-	< 1	< 1	< 1 ($N = 10$) 8 ($N = 12$) 242 ($N = 14$) > 600 ($N = 16$)
N-Queens symmetry	- (-)	- (3)	- (-)	- (6)	< 1	< 1	< 1 ($N = 10$) 6 ($N = 12$) 182 ($N = 14$) > 600 ($N = 16$)

Table 1. Systems on N-Queens related problems. All times in seconds. An entry "-" means "no solution found within 100 seconds". *N-Queens symmetry:* entries in parenthesis "($\cdot$)" refer to "tweaked" problem formulations, with redundant axioms; *Index refinement:* ($\Leftrightarrow$): proof obligation is equivalence; ($\Leftarrow$) and ($\Rightarrow$): one direction only. E, E-Darwin and Vampire can't prove the latter either.

3.2 Puzzle

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

> Five couples 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 married twice as long ago as the Watsons, but half as long as Russell.
> 3. The Shearers married ten years before Peter and ten years after 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 married most recently.
> 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, and is left for the reader's amusement. 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, the derivation is beyond the abilities of the prover.

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.

3.3 Radiation

The background for this example was given in [BBBS07], which considers the problem of decomposing an integer matrix into a positively weighted sum of binary matrices that have the so-called consecutive-ones property. We do not need the details of this problems here. Instead it suffices to say that the problem is well-known and of practical relevance. It has an important application in cancer radiation therapy treatment planning: the sequencing of multileaf collimators to deliver a given radiation intensity matrix, representing (a component of) the treatment plan.

The proof task here is along the lines as described in Section 2.6 above. It requires to show that an occurrence of the “max” function between integers can be replaced by stipulating the existence of a lower bound instead. This leads to more efficient constraint solving. The solutions are preserved, essentially, because the objective is to compute *minimal* solutions, and maxima and upper bounds leading to minimal solutions coincide then (in this example).

Besides having to prove that minimal solutions are preserved, an additional complication comes from a rather syntactically deep embedding of the max function in the constraint model. Furthermore, it occurs within a summation formula, and this way stands for a *parametric* number n, the summation bound, of usages. In addition, it occurs within a predicate definition, whose arguments are unary arrays, and the predicate is “invoked” by taking sub-arrays of certain globally defined non-unary arrays. Because it is a non-trivial exercise already to recast this constraint model in a predicate logic formula we started with a coarse abstraction of the model. We defined five proof tasks, whose differences are shown in the following table:

Problem#	**model_max**$(x, y, n) \Leftrightarrow$	**model_ub**$(x, y, n) \Leftrightarrow$
(1)	$n = \max(x, y)$	$\mathrm{ub}(x, y, n)$
(2)	$\max(x, y) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \land z \leq n)$
(3)	$\mathrm{sum}(\max(x, y)) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \land \mathrm{sum}(z) \leq n)$
(4)	$\mathrm{c} + \max(x, y) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \land \mathrm{c} + z \leq n)$
(5)	$\mathrm{c} + \mathrm{sum}(\max(x, y)) \leq n$	$\exists z\ (\mathrm{ub}(x, y, z) \land \mathrm{c} + \mathrm{sum}(z) \leq n)$

The second and third column define different abstractions of the original constraint model, models in terms of "max" and of "upper bound", respectively. It is not difficult to define minimal solutions. For the "max" version, for instance, one defines:

$$\forall x, y, z\ \text{minsol\_model\_max}(x, y, n) \Leftrightarrow \\ (\text{model\_max}(x, y, n) \land \forall z\ (\text{model\_max}(x, y, z) \Rightarrow n \leq z))\ .$$

The definition for "minsol_model_ub(x, y, n)", the minimal solutions in terms of upper bounds, is given analogously. The proof task then is to show that together with a (straightforward) axiomatization of "max" and "ub", and possibly more axioms, the equivalence

$$\forall x, y, z\ (\text{minsol\_model\_max}(x, y, n) \Leftrightarrow \text{minsol\_model\_ub}(x, y, n))$$

follows. Table 3.3 contains the results.

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. Reasoning modulo a background theory, which originated with the introduction of theory resolution, is the obvious starting point, but is it enough?

Problem		E	E-Darwin	SPASS	YICES	CVC3
	$\Leftrightarrow$	13	7	1	-	-
(1)	$\Rightarrow$	2	4	1	-	-
	$\Leftarrow$	51	2	1	1	1
	$\Leftrightarrow$	-	-	-	-	-
(2)	$\Rightarrow$	-	-	2	-	-
	$\Leftarrow$	-	-	2	-	-
	$\Leftrightarrow$	-	-	-	-	-
(3)	$\Rightarrow$	-	-	23	-	-
	$\Leftarrow$	-	-	-	-	-
	$\Leftrightarrow$	-	-	-	-	-
(4)	$\Rightarrow$	-	-	2	-	-
	$\Leftarrow$	-	-	-	-	-
	$\Leftrightarrow$	-	-	-	-	-
(5)	$\Rightarrow$	-	-	-	-	-
	$\Leftarrow$	-	-	-	-	-

Table 2. Systems on abstractions of the radiation problem. All times in seconds. $\Leftrightarrow$: proof obligation is equivalence, as stated above; $\Leftarrow$ and $\Rightarrow$: one direction only. An entry "-" means "no solution found within 100 seconds".

3. Many-sorted logic is absolutely required. There are theorem provers able to exploit sorts, but despite decades of literature on the subject, many still 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 paramodulation inferences. Despite some efforts, the state of the art in proof presentation remains unsatisfactory. This *must* be addressed somehow.[4]

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. Constraint-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

[4] The IPV tool associated with TPTP marks a good recent step towards addressing it. More in the same style needs to be the object of wider research: any serious theorem prover should routinely come with an advanced proof presentation package.

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.[5]

References

[Ano] Anonymous. Anniversaries: Logic puzzle. `http://genealogyworldwide.com/genealogy_fun.php`.

[BBBS07] Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter J. Stuckey. Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In Pascal Van Hentenryck and Laurence A. Wolsey, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings*, volume 4510 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[BCM04] Lucas Bordeaux, Marco Cadoli, and Toni Mancini. Exploiting fixable, removable, and implied values in constraint satisfaction problems. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2004.

[BFT08] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. ME(LIA) – Model Evolution With Linear Integer Arithmetic Constraints. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 258–273. Springer, November 2008.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[CM04] Marco Cadoli and Toni Mancini. Exploiting functional dependencies in declarative problem specifications. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 628–640. Springer, 2004.

[CM05] Marco Cadoli and Toni Mancini. Using a theorem prover for reasoning on constraint problems. In *AI*IA*, pages 38–49, 2005.

[DC03] Rina Dechter and David Cohen. *Constraint Processing.* Morgan Kaufmann, 2003.

[FGJ$^+$07] 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.

[5] This research was supported by NICTA (National ICT Australia) and by the Australian National University. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

[FGK02] Robert Fourer, David Gay, , and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002. `http://www.ampl.com/`.

[FPg04] 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.

[Hen99] Pascal Van Hentenryck. *The OPL optimization programming language.* MIT Press, Cambridge, MA, 1999.

[KV07] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic (CSL'07)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[MA88] Todd Feil Marlow Anderson. *Lattice-ordered Groups: An Introduction.* Springer-Verlag, 1988.

[MC05] Toni Mancini and Marco Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In Jean-Daniel Zucker and Lorenza Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2005.

[McC] William McCune. Prover9. `http://www.cs.unm.edu/ mccune/mace4/`.

[MKB+05] Darko Marinov, Sarfraz Khurshid, Suhabe Bugrara, Lintao Zhang, and Martin Rinard. Optimizations for compiling declarative models into boolean formulas. In *Computer Aided Verification (CAV)*, LNAI. Springer, 2005.

[MNR+08] 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.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[RT06] Silvio Ranise and Cesare Tinelli. Satisfiability modulo theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[Rüm08] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 274–289. Springer, November 2008.

[Wal01] Uwe Waldmann. Superposition and chaining for totally ordered divisible abelian groups. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, pages 226–241, 2001.

[WP06] Uwe Waldmann and Virgile Prevosto. Spass+t. In Stephan Schulz Geoff Sutcliffe, Renate Schmidt, editor, *ESCoREmpirically Successful Computerized Reasoning*, CEUR Workshop Proceedings, pages 18–33, Seattle, WA, USA, 2006.