Reasoning about Constraint Models

Christian Bessiere[1], Emmanuel Hebrard[2], George Katsirelos[3], Zeynep Kiziltan[4], Nina Narodytska[5], and Toby Walsh[6]

[1] CNRS and University of Montpellier, [2] LAAS-CNRS, [3] INRA Toulouse, [4] University of Bologna, [5] University of Toronto, [6] NICTA and UNSW

Abstract. We propose a simple but powerful framework for reasoning about properties of models specified in languages like AMPL, OPL, Zinc or Essence. Using this framework, we prove that reasoning problems like detecting symmetries, redundant constraints or dualities between models are undecidable even for a very limited modelling language that only generates simple problem instances. To provide tools to help the human modeller (for example, to identify when a model has a particular symmetry), it would nevertheless be useful to automate many of these reasoning tasks. To explore the possibility of doing this, we describe two case-studies. The first uses the ACL2 inductive prover to prove inductively that a model contains a symmetry. The second identifies a tractable fragment of MiniZinc and uses a decision procedure to prove that a model implies a parameterized constraint.

1 Introduction

Many modelling languages in constraint and linear programming distinguish between the model and the problem data. A model is an abstract specification of the problem, that is typically good for months or years. By comparison, the problem data is typically specific to today's problem. This can be summarised with the slogan:

$$INSTANCE = MODEL + DATA$$

For example, a model of a rostering problem might contain generic constraints about shift patterns and staffing levels. However, we still need to add problem data to specify worker availability and skills. In a modelling language like AMPL or Zinc, the model and data are in separate files. This clear separation has a number of advantages. First, models are re-usable. Every day we collect new data regarding worker availability but we can re-use the same model over a long period of time. It is therefore worth investing a lot of effort in building a good model. Second, models are more compact and easy to debug. We don't complicate the specification of the model with instance specific details. Third, models are often sufficiently abstract that we can run them in many different solvers. A modelling language is in effect a good API for interfacing solvers.

A significant challenge then is to provide tools to help the human modeller prepare good models. For example, can we automatically identify symmetries in a model? Can we automatically reformulate a model to give a dual viewpoint? A number of tools are being developed to provide such help (see Section 8 for more details). As with language

compilers, the goal is to develop powerful tools that take simple abstract models and produce sophisticated problem instances to which solvers can find solutions quickly. Unfortunately, little is known about the problem of reasoning about models. In this paper, we develop a simple theoretical framework for studying different forms of reasoning about models. We show that many reasoning tasks on models are undecidable, even with very simple models. For instance, we argue that it is undecidable to identify symmetries in a model which compiles down to very simple problem instances like clauses. Undecidability doesn't mean that these reasoning tasks cannot be partially automated. To explore this possibility, we use an automatic theorem prover to reason about models. Alternatively, we may be able to restrict the class of models and questions to a decidable language. We also explore this possibility.

2 Formal background

We want to reason about models in general. Rather than pick a specific modelling language, we make a modest assumption: the modelling language is rich enough to generate problem instances which cover any possible set of clauses. All modelling languages of which we are aware (AMPL, OPL, Zinc, MiniZinc, FlatZinc, Essence, DIMACS CNF format, and others) satisfy this assumption. Our results thus are general and not tied to a particular modelling language.

We also want to reason about models separate from the data. In many settings, the data includes the problem size. For example, the car sequencing problem (prob001 in CSPLib.org [1]) is parameterised by the number of cars on the production line. To make life simple, we suppose that the data comes in the form of a *single* parameter. A single input parameter can, however encode any number of input parameters. For example, consider the invertible mapping $m : \mathbb{N}^2 \rightarrow \mathbb{N}$, such as $m(x, y) = 2^x * 3^y$. By the Unique Prime Factorization theorem, this is a 1 to 1 mapping. Such a mapping can be extended to more than 2 parameters. For instance, n parameters $(p_1, \ldots, p_n)$ can be mapped using $p = m(n, m(p_1, m(\ldots, m(p_{n-1}, p_n)))$. Since this mapping is invertible, all parameters can be recovered from p supposing we know in advance how many parameters there are to recover.

In general, we could define a model as an arbitrary computable function. However, such a definition is too general, and all interesting questions are immediately undecidable by Rice's theorem. We instead consider a definition which matches current practice: a model is an abstract description of an instance which is *grounded* (*flattened*) to produce an actual instance. Formally we define a model as a function $M : \mathbb{N} \rightarrow S$ which generates a grounded (propositional) instance $M(p)$ where p is an appropriate encoding of the universe. We assume that S is a superset of CNF, 3-CNF or ILP. All these are equivalent since they can model any problem in NP and naturally also capture languages like AMPL, DIMACS CNF, flatzinc, Essence, NP-SPEC and others. Given parameter p, we write $M(p)$ for the problem instance generated from the model M given the data p. We will write $sol(M(p))$ for the solutions of this problem instance.

In our proofs, we will need to construct models that are the conjunction of two smaller models. We will also need to be able to force a variable in the model to take a particular value, as well as to build a model that is always unsatisfiable whatever data

is provided. All of these operations are available in modelling languages like AMPL, OPL and Zinc. They are also consistent with our assumption that our modelling language can generate any possible clause. Modelling languages may deal with decision variables in different ways. Again, our framework makes modest assumptions regarding decision variables. We suppose, for instance, that finite domains are encoded into literals constructed via the direct or order encoding.

3 Reasoning about models

There are many different reasoning tasks that we might want to perform on the parameterised model of a problem. We give here some of the more important tasks. We begin with two questions which are useful when debugging a model.

Problem 1 (Model unsatisfiability)
Given: model M.
Question: for all parameters p, is $M(p)$ unsatisfiable?

Problem 2 (Model satisfiability)
Given: model M.
Question: for all parameters p, is $M(p)$ satisfiable?

Next are two questions which are useful when comparing two models. We might want to check that two models always return the same truth value or the same solutions.

Problem 3 (Model equisatisfiability)
Given: models M_1 and M_2.
Question: for all parameters p, is it the case that $M_1(p)$ is satisfiable iff $M_2(p)$ is also?

Problem 4 (Model equivalence)
Given: models M_1 and M_2.
Question: for all parameters p, is $sol(M_1(p)) = sol(M_2(p))$?

Model equivalence implies model equisatisfiability but not vice versa. For example, a model is equisatisfiable to its dual (in the sense used in the CSP literature [2–4]) but is not equivalent. Another useful question is whether a model contains redundant constraints [5]. We might want to check if a set of constraints can be safely removed.

Problem 5 (Model redundancy)
Given: model $M = M_1 \cup M_2$.
Question: for all parameters p, is it the case that $M(p)$ is satisfiable iff $M_1(p)$ is also? Equivalently, are M and M_1 equisatisfiable?

Problem 6 (Strong model redundancy)
Given: model $M = M_1 \cup M_2$.
Question: for all parameters p, does $sol(M(p)) = sol(M_1(p))$?

We might also want to reason about the symmetries in a model. As is common, we distinguish here between variable and value symmetries. Variable symmetries are solution preserving bijections on the variable indices, whilst value symmetries are solution preserving bijections on the values [6, 7]. Value symmetries have been shown to be easier to reason with than variable symmetries. For example, we can break all value symmetries in a problem instance in polynomial time [8–10], but breaking all variable symmetries is NP-hard [11–14].

Problem 7 (Model value symmetry)
Given: model M and permutation on values θ.
Question: for all parameters p, does $M(p)$ have the value solution symmetry θ? That is, for all parameters p, does $sol(M(p)) = \theta(sol(M(p)))$?

Problem 8 (Model variable symmetry)
Given: model M and permutation on variables σ.
Question: for all parameters p, does $M(p)$ have the variable solution symmetry σ? That is, for all parameters p, does $sol(M(p)) = \sigma(sol(M(p)))$?

Another form of symmetry is symmetry *within* a solution. Such symmetries have been shown to be useful in solving a range of combinatorial problems [15]. Open problems about Van der Waerden numbers (an important concept in Ramsey theory) and about graceful graphs have been solved by developing constraint models that are restricted to have particular symmetries within their solutions.

Problem 9 (Model solution symmetry)

- *Given: Model M, and a permutation σ on assignments.*
- *Question: for all parameters p, does $M(p)$ have a symmetry σ within one of its solutions? That is, for all parameters p does there exist $sol \in sol(M(p))$ with $sol = \sigma(sol)$?*

There are often several viewpoints of a problem [2–4, 16]. For instance, with a permutation problem like the car sequencing problem, one model represents the cars by variables, and their positions in the production line by values. A dual viewpoint represents the cars by values, and the positions by variables. We might want to check if two models are exact duals of each other.

Problem 10 (Model duality)
Given: models M_1 and M_2.
Question: for all parameters p, does there exists a bijection π on complete assignments such that $sol(M_1(p)) = \pi(sol(M_2(p)))$.

Finally, to illustrate the ease with which we reason about different questions, we consider one last question. Does a model contain a backbone assignment? The backbone are those variables which are forced to the same value in all satisfying assignments. The complexity of solving a problem is typically correlated with the size of its backbone [17, 18]. In addition, branching heuristics have been developed that try to branch on backbone variables [19, 20].

Problem 11 (Model backbone)
Given: model M and an assignment $X = v$.
Question: for all parameters p and $sol \in sol(M(p))$, does $X = v \in sol$?

4 Theoretical results

We now consider how difficult it is to reason in general about these 11 problems. Our results follow from the following lemma.

Lemma 1. *Given a Turing maching T and input x, there exists a model $M_{T,x}$ that generates a satisfiable CNF instance for input n iff T terminates on x in n steps.*

The proof follows the construction used in proving that the satisfiability of CNF is NP-complete where we simulate a Turing machine with CNF clauses. This lemma demonstrates that, even under the modest assumption our modelling language can generate clauses, testing whether there exists a satisfiable grounding is undecidable. We are now in a position to state our main technical result. Even if our modelling language compiles down to a simple constraint language like CNF, all of the problems defined so far are undecidable in general.

Theorem 1 *Problems 1–11 are undecidable for a modelling language meeting our assumptions.*

Proof: For the following problems, let M be a model.

1. *Unsatisfiability*. A model always generates unsatisfiable instances if and only if it is not satisfiable, hence this follows from lemma 1.
2. *Satisfiability*. A model always generates satisfiable instances if and only if its negation is not satisfiable, so that also follows from lemma 1.
3. *Equisatisfiability*. Let $M_1(n) = M(n)$ and $M_2(n) = false$, i.e. the model that always generates an unsatisfiable instance. The models are equisatisfiable iff M is unsatisfiable.
4. *Equivalence*. As above.
5. *Redundancy*. Let $M_1(n) = M(n)$ and $M_2(n) = false$. Then $M_2(n)$ is redundant iff M is unsatisfiable.
6. *Strong redundancy*. As above.
7. *Value symmetry*. Let $M'(n) = M(n) \wedge Z = a$ for some fresh variable Z and value a. Then, for some other value b, all instances of M' have the value symmetry $\theta = (ab)$ iff M' is unsatisfiable.
8. *Variable symmetry*. Let $M'(n) = M(n) \wedge Z = 0 \wedge Y = 1$ for fresh 0/1 variables Z and Y. Then, all instances generated by M' have the variable symmetry $\sigma = (YZ)$ iff M is unsatisfiable.
9. *Solution symmetry*. Let $M'(n) = M(n) \wedge Z = 0 \wedge Y = 0$ where Z and Y are fresh 0/1 variables. We let σ be the symmetry (YZ) that swaps Y with Z. Then σ is a symmetry within a solution for all n iff M is satisfiable.

10. *Duality*. Let $M_1(n) = M(n) \wedge (Z = 0 \vee Z = 1)$ and $M_2(n) = M(n) \wedge Z = 0$ where Z is a fresh 0/1 variable. If $M(n)$ is satisfiable, then as $M_1(n)$ has twice the solutions (i.e. all those with $Z = 0$ like $M_2(n)$ but also an equal number with $Z = 1$) there can never be a bijection between the solutions of $M_1(n)$ and $M_2(n)$. Hence, $M_1(n)$ and $M_2(n)$ are only dual iff M is unsatisfiable.
11. *Backbone.* We construct a formula M' by adding the propositional literal $\neg X$ to every clause of M, where X is a fresh variable. If M is satisfiable, then any solution of $M(n)$ can be extended to a solution of $M'(n)$ either with $X = 0$ or $X = 1$. Otherwise, if M is unsatisfiable, a refutation of $M(n)$ can be transformed to a derivation of the clause $(\neg X)$ in $M'(n)$. Moreover, setting $X = 0$ satisfies all clauses, so $X = 0$ is in the backbone of $M'(n)$ if and only if M is unsatisfiable. □

We make some observations about these results. First, it is perhaps not too surprising that many of these problems are undecidable. Modelling languages can express very rich problems and deciding, for example, if two models are dual is clearly a non-trivial task since it requires reasoning about the set of models of every problem instance. However, what may be surprising is that we keep this undecidability when the instances generated by models are very simple. In particular, all of these reasoning problems are undecidable when restricted to models that generate just clauses. Second, some differences in difficulty at the instance level disappear at the model level. For example, at the instance level, value symmetries are easier to reason with than variable symmetries. We can break all value symmetries in a given problem instance in polynomial time [8], but breaking all variable symmetries is NP-hard [11]. However, at the model level, value symmetries appear to be just as difficult to reason with as variable symmetries. Third, we can show that many other reasoning tasks are undecidable under similar assumptions. For example, we can give similar proofs of undecidability for reasoning about other properties like the interchangeability of values. We are, however, limited to properties that can be expressed in terms of the solutions of an instance, rather than *structural* properties of an instance. It is not clear how to reason about the decidability of problems like the following.

Problem 12 (Bounded tree-width)
Given: model M and integer k.
Question: for all parameters p, does $M(p)$ have tree-width $\leq k$?

5 A simple example

To illustrate how easy it is to run into undecidability, we give a small MiniZinc model that is undecidable to reason about. Given seven 3 by 3 matrices of integers, M_1 to M_7 it is undecidable to determine if there exists a sequence $S_1, \ldots, S_n$ of M_2 to M_7 (perhaps containing repeated entries) which yields the matrix product $M = S_1 \cdot \ldots \cdot S_n$ in which the top left entry of $M_1 M$ is zero [21]. Here is a MiniZinc model for this problem:

```
int: n;
array [1..3,1..3,1..7] of int: data;
array [1..n-1] of var 2..7: seq;
```

```
array [1..3,1..3,1..n] of var int: prod;
constraint
 forall(i in 1..3)
  (forall(j in 1..3)
   (prod[i,j,1] = data[i,j,1] /\
    forall(k in 2..n)
     (prod[i,j,k] =
      sum([prod[i,l,k-1]*data[l,j,seq[k]]
             | l in 1..3]))
    )
   )
  );
constraint  prod[1,1,n]=0;
solve satisfy;
```

Given input data for the seven matrices, it will be undecidable to determine if there are any satisfiable instances of this model for the parameter n. We can also use this model as a basis of models that demonstrate the undecidability of checking all 11 problems defined so far like model satisfiability, model equivalence, etc. We would just need to repeat the constructions used in the proof of Theorem 1 with this model of matrix multiplication.

6 Inductively reasoning about models

Although it is undecidable to check if a model has a property like redundant constraints or a given symmetry, we can nevertheless use heuristic methods to decide many cases. We may be willing to invest considerable effort in reasoning about a model as the cost is amortized over many instances. For example, we might use an automated theorem prover to prove that a model implies a certain implied constraint. Here we consider the ACL2 prover which is one of the most powerful inductive provers developed to date. Induction permits us to reason about all possible input parameters. We illustrate the approach with a simple example of proving that a model is reversal symmetric. The all interval series problem (prob007 in CSPLib [1]) asks for a permutation of the numbers 0 to $n-1$ so that neighbouring differences form a permutation of 1 to $n-1$. We model this as a constraint satisfaction problem in n variables with the ith variable equal to j iff the ith number in the series is j. One symmetry of this model is that we can reverse the series and construct another solution. As ACL2 constructs inductive proofs about recursively defined functions, we first write down a recursive model of our constraints and the neighbouring difference function. It is relatively straight forward to extract such recursive functions from, say, a constraint logic programming model of the problem. We begin with definitions of three functions: `domain` that checks the domain of a series of variables, `alldiff` that ensures that a series of variables are pairwise different, and `diffs` that returns the absolute difference between neighbouring variables in a series.

```
(defun domain (series lo hi)
  (if (endp series) T
      (and (<= lo (car series))
           (>= hi (car series))
           (domain (cdr series) lo hi))))
```

```
(defun alldiff (series)
  (if (endp series) T
      (and (not (member (car series)
                        (cdr series)))
           (alldiff (cdr series)))))
(defun diffs (series)
  (if (endp series) nil
      (if (endp (cdr series)) nil
          (cons (abs (- (car series)
                        (car (cdr series))))
                (diffs (cdr series))))))
```

Using these three functions, we can now model the problem with a simple conjunction, $\varphi(series)$ as follows:

```
(and
  (listp series) (domain series 0 n)
  (alldiff series) (alldiff (diffs series)))
```

We formalize the proof obligation that the model contains the reversal symmetry group as follows:

```
(thm
 (implies
  (and (listp series) (domain series 0 n)
       (alldiff series) (alldiff (diffs series)))
  (and (listp (rev series))
       (domain (rev series) 0 n)
       (alldiff (rev series))
       (alldiff (diffs (rev series))))))
```

Unfortunately, ACL2 will not prove this theorem without assistance based on the definitions alone. We also need to prove a few standard rewrite rules about combinations of constraints and functions that appear in the theorem:

```
(defthm mem-app
  (iff (member-equal x
           (append series1 series2))
       (or (member-equal x series1)
           (member-equal x series2))))
(defthm mem-rev
  (iff (member-equal n (rev series))
       (member-equal n series)))
(defthm alldiff-app
  (implies (and (not (member-equal x series))
                (alldiff series))
           (alldiff (append series (list x)))))
(defthm diffs-app
  (equal
     (diffs (append x y))
     (cond ((endp x) (diffs y))
           ((endp y) (diffs x))
```

```
           (t (append (diffs x)
                (cons (abs (- (car (last x))
                              (car y)))
                      (diffs y)))))))
(defthm car-last-rev
  (equal (car (last (rev x)))
         (car x)))
(defthm diffs-rev
  (equal (diffs (rev series))
         (rev (diffs series))))
```

With these rewrite rules, ACL2 is able to prove automatically the existence of the reversal symmetry. The proof takes 20.44 seconds on a 2.53 GHz Intel Core i5 with 4GB RAM. The proof has 697,643 steps. Many of the rewrite rules introduced to get the prover to work automatically like `mem-rev` and `alldiff-app` are likely to be useful to reason about other models containing reversal symmetries and all different constraints. It might therefore be possible to build background theories of rewrite rules with which many properties of models could be automatically proved. Nevertheless, the size of the proof and the number of additional rewrite rules needed even for this relatively simple problem demonstrate that automatic theorem provers are likely to be challenged by these sort of reasoning tasks. We therefore consider a more automated alternative.

7 Tractable Languages

A simpler possibility is to restrict the models and questions we ask to a decidable language. Consider the fragment $\mathcal{L}$ of MiniZinc with the following restrictions: there are a fixed number of integer or set variables (with the set variables limited to a fixed universe of values), integer parameters, quantification, sums and set comprehension of bounded size (so we can unroll all such expressions), linear equality and inequality constraints, set constraints (made from predicates like $\subseteq$ and $=$, as well as function like $\cup$ and $\cap$), any of the MiniZinc global constraints (including *alldifferent*, $among$, $atmost$, $binpacking$, $cumulative$, and $element$), Boolean combinations of any of the previously mentioned constraints (including reification of these constraints), and either a satisfaction goal or a maximization or minimization goal involving an integer linear expression. Integer parameters in such models can represent input data (e.g. demands), as well as variable domains of unbounded size.

Theorem 2 *Any model and question that can be formulated in $\mathcal{L}$ is decidable.*

Proof: We exploit the fact that it is decidable to reason about quantified Presburger arithmetic. We have to demonstrate that all constraints in $\mathcal{L}$ can be expanded into logically equivalent expressions in linear integer arithmetic. For global constraints, we transform each into a Boolean combination of linear arithmetic constraints. For example, $atmost(N, [X_1, \ldots, X_m], v)$ is transformed into $X_1 = v \leftrightarrow S_1 = 1$, $X_1 \neq v \leftrightarrow S_1 = 0$, $X_i = v \leftrightarrow S_i = 1 + S_{i-1}$, $X_i \neq v \leftrightarrow S_i = S_{i-1}$ where $1 < i \leq m$ and $N \leq S_m$.

For set constraints, by introducing set variables for intermediate expressions, we can replace nested set expressions with binary or ternary set constraints (e.g. $(A \cap B) \cup C = D$ can be transformed to $AB = A \cap B$, $D = AB \cup C$). Such binary and ternary set constraints can themselves be rewritten as Boolean combinations of equalities and inequalities over 0/1 variables representing the characteristic function of the set (e.g. $A \cap B = C$ can be rewritten $\forall i \in U.(A_i \wedge B_i) \leftrightarrow C_i$ where U is the fixed universe of possible set values).

A satisfaction goal is represented simply by the conjumction of the transformed linear arithmetical constraints. A maximization (dually minimization) goal is represented by a more complex expression. For instance, to maximize $\sum_i w_i.X_i$ subject to the constraints $C(X_1, \ldots, X_n)$, we construct the formula $\forall X_1, Y_1, \ldots, X_n, Y_n.C(X_1, \ldots, X_n) \wedge C(Y_1, \ldots, Y_n) \rightarrow \sum_i w_i.Y_i \leq \sum_i w_i.X_i$. Finally integer parameters are replaced by universally quantified integer variables, and decision variables by existentially quantified integer variables. □

It is decidable to ask any questions about such models that can themselves be expressed in this fragment $\mathcal{L}$ of MiniZinc. As a simple case-study, we consider a simple MiniZinc model of the 3 by 3 magic square problem, parameterized by the magic number m. A magic square is a n by n square of distinct integers such that each row, column and diagonal sum to the same magic number m (prob019 in CSPLib). Note that we are not restricting ourselves to *normal* magic squares where the square needs to contain just the integers 1 to n^2 and m is fixed. We consider magic squares which contain any set of n^2 distinct integers and m is not fixed.

There are an infinite number of 3 by 3 magic squares including the only normal one, the ancient Chinese *Lo Shu* square:

4	9	2
3	5	7
8	1	6

There are even magic square of primes, like this one containing Chen primes discovered by Rudolf Ondrejka:

17	89	71
113	59	5
47	29	101

It is easy to see that this can be modelled in MiniZinc by linear (in)equalities over a fixed number of variables. As a case study, we used a Presburger arithmetic decision procedure to prove two constraints implied by a parameterized MiniZinc model of the 3 by 3 magic squares problem. First, we proved that the middle square always takes the average value $m/3$. For example, in the magic square of Chen primes introduced above, $(17 + 89 + 71)/3 = (113 + 59 + 5)/3 = (47 + 29 + 101)/3 = 177/3 = 59$. Such an implied constraint will reduce search greatly as it removes a variable from the problem. ACL2's decision procedure for integer linear arithmetic is able to prove this statement in 0.01 seconds, taking just 310 proof steps. Note that the implied constraint proven to hold (that is, $3x_{2,2} = m$) contains the model's parameter (m). Such an implied constraint cannot be shown to hold using a constraint solver as we need to prove it is true for all

satisfying assignments of each of the (potentially infinite) values of m. We also proved another implied constraint: the four corner squares of the magic square always have the same sum as the four cardinal squares. For example, in the magic square of Chen primes introduced earlier, $17 + 71 + 101 + 47 = 89 + 5 + 29 + 113 = 236$. ACL2's decision procedure for integer linear arithmetic is again able to prove this statement taking just 0.01 seconds, and 330 proof steps. This cannot be shown using a constraint solver as we need to prove it holds for all values of m. This case study demonstrates considerable potential for using decision procedures.

8 Related work

There has been some prior work on reasoning about constraint models. For example, NP-SPEC uses existential second-order logic as a modelling language in which we can express models of any NP-complete problem [22]. Cadoli and Mancini have argued that detecting value symmetries is undecidable in this modelling language [23]. Unfortunately this proof has never been published though we were able to obtain it privately from the second author. This proof uses a reduction from checking if an arbitrary closed first-order formula is a tautology (which is undecidable). The undecidability of detecting value symmetries in NP-SPEC models follows directly from Theorem 1 as NP-SPEC meets the weak assumptions of the theorem. Cadoli and Mancini also demonstrated that value symmetries can be effectively detected on some simple models like that of the social golfers problem (prob010 in CSPLib [1]) using the OTTER first order theorem prover and the MACE finite model finder in fully automatic modes [24]. More recently, Slaney and Baumgartner have demonstrated promise in using SMT solvers to reason about the existence of symmetries and of implied constraints in simple models of the n-queens, logic puzzles and radiation therapy problems [25].

Mears *et al.* have detected symmetries on small instances of a constraint model, and used this to identify symmetries that might occur in the more general model [26]. More recently, Mears *et al.* have shown how a limited form of symmetry detection in MiniZinc can be reduced to deciding sentences in Presburger arithmetic [27]. Their method is more limited than that explored in our case studies as they use Presburger arithmetic simply to identify symmetries that map the indices on one constraint onto the indices in another. Despite this restriction, they were able to find some matrix symmetries in models of Latin squares, social golfers and other simple combinatorial problems. They also argued that detecting value and variable symmetry in MiniZinc matrix models is undecidable in general based on the reduction from an undecidable tiling problem. There are a number of differences with the undecidability results present here. First, our results are not limited to MiniZinc but to a wide range of modelling languages. Second, our results are not limited to reasoning just about symmetries. We reason about many other properties of models like duality and redundancy. Third, our proof is directly from the Halting problem rather than indirectly through grid tiling.

Modern modelling languages are designed to be compiled down into efficient problem instances. For example, ESSENCE is a high level modelling language that is compiled down into problem instances using the CONJURE rewrite rule system [28]. A major component of CONJURE is the choice of representation for abstract data types

like set and multiset variables [29]. However, a future goal is to reason about implied constraints and symmetry breaking constraints that can be added as the model is refined down to a problem instance.

A number of techniques and tools have been developed to reason about problem instances. For example, Puget has shown that symmetries of a problem instance can be detected by finding automorphisms of a suitable coloured graph [30]. He has also extended the method to deal with a limited class of logical and arithmetical constraints specified intensionally. As a second example, Charnley, Colton and Miguel built a system to conjecture implied constraints from solutions to small instances of a problem [31]. The first order theorem prover OTTER was then used to attempt to prove that these constraints are implied. As a third example, Rossi, Petrie and Dhar consider the problem of deciding when two instances of a constraint problem are "equivalent" [32]. They argue that having the same set of solutions is often inadequate (model equivalence in our more general setting), and that a notion of reducibility in which solutions of one instance can be obtained from solutions of the other and vice versa (model duality in our more general setting). They use this framework to prove the equivalence of (instances of) binary and non-binary constraint satisfaction problems.

9 Conclusions

We have proposed a simple but powerful framework for reasoning about properties of modelling languages like AMPL, OPL, Zinc, Essence and NP-SPEC. Using this framework, we have shown that even for a very limited modelling language many properties like deciding satisfiability of model or redundancy of constraints are undecidable. Despite this undecidability, it is interesting to attempt to automate many of these reasoning tasks so we can provide tools that help the human modeller. For example, we might want to identify when a user adds inconsistent constraints into their model or when a model contains a particular symmetry. To explore the possibility of providing such automation, we describe two simple case-studies. The first used the ACL2 inductive prover to prove inductively that a model of the all interval series problem contains a reversal symmetry. The second identified a tractable fragment of MiniZinc which can be decided using Presburger arithmetic. We illustrated this by proving that a model of the magic square problem implies certain (parameterized) constraints. Our results justify why heuristic and incomplete methods have been proposed previously to automate tasks like identifying symmetries or implied constraints.

Acknowledgements

NICTA is supported by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. The last author is currently supported by the Federal Ministry for Education and Research through the Alexander von Humboldt Foundation, as well as by AOARD Grant FA2386-12-1-4056.

References

1. Gent, I., Walsh, T.: CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999 (1999) A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).
2. Geelen, P.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proceedings of the 10th ECAI, European Conference on Artificial Intelligence (1992) 31–35
3. Cheng, B., Choi, K., Lee, J., Wu, J.: Increasing constraint propagation by redundant modeling: an experience report. Constraints **4** (1999) 167–192
4. Walsh, T.: Permutation problems and channelling constraints. In: Proceedings of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001). (2001)
5. Smith, B., Stergiou, K., Walsh, T.: Using auxiliary variables and implied constraints to model non-binary problems. In: Proceedings of the 16th National Conference on AI, Association for Advancement of Artificial Intelligence (2000) 182–187
6. Gent, I., Petrie, K., Puget, J.F.: Symmetry in constraint programming. In: Handbook for Constraint Programming. Elsevier (2006)
7. Walsh, T.: General symmetry breaking constraints. In Benhamou, F., ed.: Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings. Volume 4204 of Lecture Notes in Computer Science., Springer (2006)
8. Puget, J.F.: Breaking all value symmetries in surjection problems. In van Beek, P., ed.: Proceedings of 11th International Conference on Principles and Practice of Constraint Programming (CP2005), Springer (2005)
9. Walsh, T.: Symmetry breaking using value precedence. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: 17th European Conference on Artificial Intelligence. Volume 141 of Frontiers in Artificial Intelligence and Applications., IOS Press (2006) 168–172
10. Walsh, T.: Breaking value symmetry. In: 13th International Conference on Principles and Practices of Constraint Programming (CP-2007), Springer-Verlag (2007)
11. Crawford, J., Ginsberg, M., Luks, G., Roy, A.: Symmetry breaking predicates for search problems. In: Proceedings of the 5th International Conference on Knowledge Representation and Reasoning, (KR '96). (1996) 148–159
12. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. In: Proceedings of the 19th National Conference on AI, Association for Advancement of Artificial Intelligence (2004)
13. Katsirelos, G., Narodytska, N., Walsh, T.: On the complexity and completeness of static constraints for breaking row and column symmetry. In Cohen, D., ed.: Proceedings of the 16th International Conference on the Principles and Practice of Constraint Programming (CP 2010). Volume 6308 of Lecture Notes in Computer Science., Springer (2010) 305–320
14. Walsh, T.: Breaking value symmetry. In Fox, D., Gomes, C., eds.: Proceedings of the 23rd National Conference on AI, Association for Advancement of Artificial Intelligence (2008) 1585–1588
15. Heule, M., Walsh, T.: Symmetry within solutions. In: Proceedings of the 24th National Conference on AI, Association for Advancement of Artificial Intelligence (2010)
16. Hnich, B., Walsh, T., Smith, B.: Dual modelling of permutation and injection problems. J. Artif. Intell. Res. (JAIR) **21** (2004) 357–391
17. Kilby, P., Slaney, J., Thiebaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Proceedings of the 20th National Conference on AI, Association for Advancement of Artificial Intelligence (2005)

18. Slaney, J., Walsh, T.: Backbones in optimization and approximation. In: Proceedings of 17th IJCAI, International Joint Conference on Artificial Intelligence (2001)
19. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hardn 3-SAT formulae. In: Proceedings of the 17th International Conference on AI, International Joint Conference on Artificial Intelligence (2001) 248–253
20. Kilby, P., Slaney, J., Walsh, T.: The backbone of the travelling salesperson. In: IJCAI. (2005) 175–180
21. Halava, V., Harju, T., Hirvensalo, M.: Undecidability bounds for integer matrices using clause instances. Int. J. Found. Comput. Sci. **18**(5) (2007) 931–948
22. Cadoli, M., Ianni, G., Palopoli, L., Schaerf, A., Vasile, D.: NP-Spec: an executable specification language for solving all problems in NP. Comput. Lang. **26**(2-4) (2000) 165–195
23. Mancini, T., Cadoli, M.: Detecting and breaking symmetries by reasoning on problem specifications. In Zucker, J.D., Saitta, L., eds.: 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005). Volume 3607 of Lecture Notes in Computer Science., Springer (2005) 165–181
24. Cadoli, M., Mancini, T.: Using a theorem prover for reasoning on constraint problems. Applied Artificial Intelligence **21**(4&5) (2007) 383–404
25. Baumgartner, P., Slaney, J.: Constraint modelling: A challenge for first order automated reasoning. In Peltier, N., Sofronie-Stokkermans, V., eds.: Proceedings of the 7th International Workshop on First-Order Theorem Proving (FTP'09). Volume 556., CEUR (2009) 4–18
26. Mears, C., de la Banda, M.G., Wallace, M., Demoen, B.: A novel approach for detecting symmetries in CSP models. In Perron, L., Trick, M., eds.: 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2008). Volume 5015 of Lecture Notes in Computer Science., Springer (2008) 158–172
27. Mears, C., Niven, T., Jackson, M., Wallace, M.: Proving symmetries by model transformation. In Lee, J.M., ed.: 17th International Conference on Principles and Practice of Constraint Programming (CP 2011). Volume 6876 of Lecture Notes in Computer Science., Springer (2011) 591–605
28. Frisch, A., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of essence: A constraint language for specifying combinatorial problems. In Veloso, M., ed.: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007). (2007) 80–87
29. Akgun, O., Miguel, I., Jefferson, C., Frisch, A., Hnich, B.: Extensible automated constraint modelling. In Burgard, W., Roth, D., eds.: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011), AAAI Press (2011)
30. Puget, J.F.: Automatic detection of variable and value symmetries. In van Beek, P., ed.: Proceedings of 11th International Conference on Principles and Practice of Constraint Programming (CP 2005). Volume 3709 of Lecture Notes in Computer Science., Springer (2005) 475–489
31. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: 17th European Conference on Artificial Intelligence (ECAI 2006). Volume 141 of Frontiers in Artificial Intelligence and Applications., IOS Press (2006) 73–77
32. Rossi, F., Petrie, C., Dhar, V.: On the equivalence of constraint satisfaction problems. In: Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90). (1990) 550–556