ECAI 2014
T. Schaub et al. (Eds.)

doi:10.3233/978-1-61499-419-0-3

Breaking Conditional Symmetry in Automated Constraint Modelling with CONJURE

Ozgur Akgun, Ian P. Gent, Christopher Jefferson, Ian Miguel and **Peter Nightingale** [1]

Abstract. Many constraint problems contain *symmetry*, which can lead to redundant search. If a partial assignment is shown to be invalid, we are wasting time if we ever consider a symmetric equivalent of it. A particularly important class of symmetries are those introduced by the constraint modelling process: *model* symmetries. We present a systematic method by which the automated constraint modelling tool CONJURE can break conditional symmetry as it enters a model during refinement. Our method extends, and is compatible with, our previous work on automated symmetry breaking in CONJURE. The result is the automatic and complete removal of model symmetries for the entire problem class represented by the input specification. This applies to arbitrarily nested conditional symmetries and represents a significant step forward for automated constraint modelling.

1 Introduction

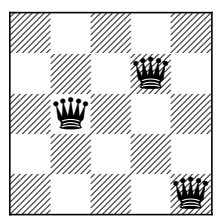
Many constraint problems contain *symmetry*. That is, given a solution to an instance we can find another symmetric solution. Symmetry can lead to redundant search. If a partial assignment is shown to be invalid, we are wasting time if we ever consider a symmetric equivalent of it. A variety of methods are available for 'symmetry breaking', i.e. avoiding reporting equivalent solutions and doing redundant search. Symmetry in constraints, and especially symmetry breaking, has been the subject of much research [17].

A particularly important class of symmetries are those introduced by the constraint modelling process: these are called *model* symmetries [13] and can occur even if the original problem has no symmetry. An example would be representing a set of size n by a vector of n constrained variables, required to be all different. Without care, this can introduce $n!$ symmetries, for the set represented by the vector in all possible orders. If the elements of the set are integers, there is no deep problem: we can add the constraint that the integers are increasing. However, this simple approach cannot be used directly if the elements of the set are themselves (for example) sets of multisets. This can lead to a dilemma. If the constraint problem is modelled at a high level, in which sets of multisets are first class objects, we may not be able to break the symmetry we introduce at the modelling level. If the problem is modelled at a low level, e.g. with all variables as integers, the resulting symmetry group may be complex and the necessary set of symmetry breaking constraints hard to specify.

Recently, we solved this dilemma in the context of our automated constraint modelling system CONJURE [1]. We generalised the approach of ordering variables by introducing a total ordering $\lesssim$ on types in CONJURE. The ordering can be used to introduce symmetry breaking constraints for symmetries that CONJURE introduces as a part of its automated modelling refinements. This is automatic, since each refinement rule indicates how to break any symmetry it introduces. This obviates the need for an expensive symmetry detection step following model formulation, as used by other approaches [23, 25]. Furthermore the symmetry breaking constraints added hold for the entire parameterised problem class captured by the ESSENCE specification — not just a single problem instance – without the need to employ a theorem prover.

In this paper we solve a major problem not addressed by our previous work. We show how CONJURE can break a different kind of symmetry: *conditional symmetry* [16]. A conditional symmetry is one which is not necessarily present in every solution: hence it is conditional on properties of the solution. To illustrate how conditional symmetry arises in constraint models, we consider the Dominating Queens problem [18], recently used at the First International Lightning Model and Solve Competition[2]:

> Given a positive integer m, minimise the number of queens placed on an $m \times m$ chess board such that no pair of queens attack each other, and every unoccupied square of the board is attacked by some queen.



The illustration shows a picture of a solution for $m = 5$ and the minimal number of 3 queens. A natural way to consider the decision being made in solving the Dominating Queens is as finding a partial function from the m rows of the chess board to the m possible positions for a queen on each row (the columns). There are several ways to model a partial function in a constraint model. A common approach is to employ a matrix, which we will call *board* in this example, of decision variables indexed by $1..m$, each of which also has the domain $\{1..m\}$. The assignment $board[i] = j$ indicates that the queen associated with the ith row is assigned to the jth column. In order to make the function partial we add a further matrix of decision variables, which we will call *switches*, also indexed by $1..m$ but with domain $\{0, 1\}$. The assignment $switches[i] = 1$ indicates that the ith row has an image in the partial function we are modelling, whereas $switches[i] = 0$ indicates that the ith row has no image, or equivalently that no queen is placed on the ith row.

This model of a partial function has conditional symmetry [16]. When $switches[i] = 0$, the values of $board[i]$ become interchangeable because the switch indicates that the ith row has no queen as-

[1] University of St Andrews, UK, email: {ozgur.akgun, ian.gent, caj21, ijm, pwn1}@st-andrews.ac.uk

[2] `http://cp2013.a4cp.org/program/competition`

signed to it. This can have serious consequences for the performance of the constraint solver in solving the model, since every dead end visited in the search can potentially have many symmetric equivalents, which will all be visited in the worst case. One approach to breaking this symmetry is to add constraints to fix the value of $board[i]$ when $switches[i] = 0$, e.g.:

$\forall i$ in $1..m$. $switches[i] = 0 \rightarrow board[i] = 1$

where we arbitrarily picked the value 1 as our "dontCare" value. As we will demonstrate, conditional symmetry arises very frequently not just in models of partial functions but also in models of other fundamental structures such as sets, multisets and relations.

To deal with model conditional symmetries, we designate each variable of each type as having a 'dontCare' value in its domain. When the condition for a given symmetry applies, we state that an affected variable must take its dontCare value. A dontCare value for an abstract decision variable does not need to satisfy the structural constraints of its domain. We show how this can be done during the refinement process of automated modelling. Furthermore, this can be done in such a way that at the final stage the dontCare values are replaced by explicit values, meaning that no adaptation is required of other tools or solvers. We describe how dontCare values are achieved and refined in sets, multisets, relations, partitions and partial functions. We show that dontCare values can be dealt with correctly in nested types. Our experimental results show that, as expected, our technique can yield greatly reduced search. Finally we give an analysis to show that dontCare values combine correctly with other symmetry breaking techniques.

2 Automated Constraint Modelling & CONJURE

The *modelling bottleneck* characterises the difficulty of formulating a problem of interest as a constraint model suitable for input to a constraint solver. The space of possible models for a given problem is typically large, and the model selected can have a dramatic effect on the efficiency of constraint solving. This presents a serious challenge for the inexpert user, who has difficulty in formulating a good (or even correct) model, and motivates efforts to automate constraint modelling. Several approaches have been taken to automate aspects of constraint modelling, some of which include: learning models from positive or negative examples [3–5, 7, 21]; automated transformation of medium-level solver-independent constraint models [26–29]; theorem proving [6]; case-based reasoning [22]; and refinement of abstract constraint specifications [11] in languages such as ESRA [10], ESSENCE [12], $\mathcal{F}$ [19] or Zinc [20, 24].

In this paper our focus is on the refinement-based approach, where a user writes *abstract* constraint specifications that describe a problem above the level at which constraint modelling decisions are made. Abstract constraint specification languages, such as ESSENCE or Zinc, support abstract decision variables with types such as set, multiset, relation and function, as well as *nested* types, such as set of sets and multiset of relations. Problems can typically be specified very concisely in this way, as demonstrated by the example in Figure 1. However, existing constraint solvers do not support these abstract decision variables directly, so abstract constraint specifications must be *refined* into concrete constraint models.

We use ESSENCE [12] herein. An ESSENCE specification, such as that in Figure 1, identifies: the input parameters of the problem class (given), whose values define an instance; the combinatorial objects to be found (find); and the constraints the objects must satisfy (such that). In addition, an objective function may be specified (min/maximising) and identifiers declared (letting). Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. Our CONJURE system[3] [2] employs refinement rules to convert an ESSENCE specification into the solver-independent constraint modelling language ESSENCE′ [28]. From ESSENCE′ we use SAVILEROW[4] to translate the model into input for a particular constraint solver while performing solver-specific model optimisations.

```
given n: int
letting ROW, COL be domain int(1..n)

find board: function (injective) ROW --> COL

minimising |board|

such that
  forAll (r1,c1),(r2,c2) in toSet(board)
    , r1 < r2 . |c1-c2| != |r1-r2|

such that
  forAll r : ROW, !(r in defined(board)) .
    forAll c : COL .
      (exists r2 : ROW , r != r2 .
        board(r2) = c) \/
      (exists r2 : ROW , r != r2 .
        |board(r2) - c| = |r2 - r|)
```

Figure 1: ESSENCE specification of the Dominating Queens Problem.

Every refinement rule that introduces conditional symmetry also generates a constraint to break those symmetries. The other symmetries introduced by refinement are independent, so we can add constraints that immediately break each introduced group of symmetries in a valid and complete manner. This leads to globally valid and complete symmetry breaking.

To illustrate how CONJURE rules can be extended to generate symmetry-breaking constraints of this form, consider the rule given below, which models a partial injective function using a matrix of tuples. The first component of the tuple indicates if the function is defined for this index value. When this first component is true, the second component gives the result of the function. A constraint is posted to ensure injectivity of the function variable: this constraint works on every distinct pair of indices and produces a disequality constraint on the second component of the tuple on the condition that the corresponding first components take the value true.

```
Name:       Function~1DPartial
Matches:    function (injective) &fr --> &to
Produces:   refn : matrix indexed by [&fr] of (bool, &to)
Constraint: forAll i,j : &fr , i != j /\ refn[i][1]
              /\ refn[j][1] . refn[i][2] != refn[j][2]
```

This rule successfully breaks the symmetry on active parts of the function domain. However, where the first component of a position in the matrix takes the value false the second component is unconstrained as its value does not affect the function being represented. This is exactly the kind of symmetry we want to break using dontCare constraints; adding the following constraint without modifying the rule fixes inactive parts of the function domain to a single value.

```
forAll i : &fr . !refn[i,1] -> dontCare(refn[i,2])
```

3 Sources of Conditional Symmetry

ESSENCE has five abstract type constructors corresponding to five of the most common combinatorial objects that combinatorial problems typically require us to find: set, multiset, relation, partition and

[3] http://bitbucket.org/stacs_cp/conjure-public
[4] http://savilerow.cs.st-andrews.ac.uk

function. Any type constructed with one (or a combination) of these must be refined before a model can be output in ESSENCE′. Conditional symmetry can arise from the refinement of *all* the abstract types formed using these constructors, as we will demonstrate.

In what follows we will show one or more refinements for each of the five type constructors listed above, each corresponding to a CONJURE refinement rule. Typically, representing an abstract domain like `set` using a more concrete domain like `matrix` requires the addition of *structural constraints* in order to maintain the invariants of the original domain, such as distinctness of members of a set. Symmetry breaking constraints are added by refinement rules in the form of additional structural constraints. The operators `.<` and `.<=` are often used to order expressions and to break symmetry. Where conditional symmetry is introduced by a refinement rule, we show the `dontCare` constraint required to break it. In Section 4, we will discuss how these `dontCare` constraints are handled.

3.1 Sets

Conditional symmetry can arise when refining sets with unknown cardinality. Consider the following set with unknown but bounded size, where τ can be any ESSENCE domain.

```
find s: set (maxSize n) of τ
```

The *explicit* refinement of `s` is shown below. In this refinement, each element in `s` is explicit in matrix `sVal`.

```
find sVal: matrix indexed by [int(1..n)] of τ
find sUsed: matrix indexed by [int(1..n)] of bool
such that
  forAll i : int(1..n-1) .
    sUsed[i+1] -> sUsed[i],
  forAll i : int(1..n-1) .
    sUsed[i+1] -> sVal[i] .< sVal[i+1]
```

Some variables in `sVal` may not be significant (when `sUsed[i]` is `false`, `sVal[i]` is not used), therefore this refinement has conditional symmetry. The following additional constraint breaks the conditional symmetry.

```
forAll i : int(1..n) . !sUsed[i] -> dontCare(sVal[i])
```

The *marker variable* refinement of `s` has a variable indicating the size of the set, as shown below.

```
find sVal: matrix indexed by [int(1..n)] of τ
find ssize: int(0..n)
such that
  forAll i : int(1..n-1) .
    i+1 <= ssize -> sVal[i] .< sVal[i+1]
```

The marker variable refinement introduces conditional symmetry when variables in `sVal` are unused. The following additional constraint breaks the conditional symmetry.

```
forAll i : int(1..n) . i > ssize -> dontCare(sVal[i])
```

Both of the above set refinements work independently of τ. The special case of τ being an integer domain can be represented without introducing conditional symmetry. CONJURE contains two refinement options for sets of integers. The first is the *dummy value* refinement which uses a value that is not in the original integer domain to indicate unused variables. The second is the *occurrence* refinement which uses a matrix of boolean variables indexed by the integer domain. These two refinements do not introduce conditional symmetry, so do not need the addition of new constraints to break it.

3.2 Multisets

The refinement of multiset domains with unknown cardinality can also introduce conditional symmetry. Consider the following multiset domain with unknown but bounded size, where τ can be any ESSENCE domain.

```
find ms : mset (maxSize n) of τ
```

CONJURE has explicit and occurrence refinements of multiset domains. These are analogous to the set refinements, with the difference being that the boolean variables are replaced with integers representing the number of occurrences of a value.

The explicit refinement models each element in the explicit matrix `msVal`.

```
find msVal: matrix indexed by [int(1..n)] of τ
find msOccur: matrix indexed by [int(1..n)] of int(0..n)
such that
  forAll i : int(1..n-1) .
    msOccur[i+1] > 0 -> msOccur[i] > 0,
  forAll i : int(1..n-1) .
    msOccur[i+1] > 0 -> msVal[i] .< msVal[i+1],
  (sum i : int(1..n) . msOccur[i]) <= n
```

The value of `msOccur` models the number of occurrences of a value. Conditional symmetry arises when `msOccur[i]` is `0`, and it can be broken using the following additional constraint.

```
forAll i : int(1..n) . msOccur[i]=0 -> dontCare(msVal[i])
```

Similar to the occurrence refinement of sets, the occurrence refinement of multisets does not introduce conditional symmetry.

3.3 Relations

ESSENCE includes relation domains of any arity, and the refinement of relations with unknown number of entries can introduce conditional symmetry. Consider a relation of arity 2 and unknown but bounded size.

```
find r : relation (maxSize n) of (τ * τ)
```

One refinement of `r` is to represent the relation as a set of tuples, then use the explicit representation of a set, as shown above. This introduces conditional symmetry because some variables are unused when the relation is smaller than its maximum size. The conditional symmetry is broken by reusing the implementation for set domains.

A second refinement of `r` uses a two-dimensional matrix of boolean variables, where each entry in the matrix represents the inclusion of one tuple in the relation. This refinement is similar to occurrence refinements of sets and multisets; it only works on integer domains but does not introduce any conditional symmetry.

3.4 Partitions

Partitions in ESSENCE are a set of non-empty, disjoint sets of values drawn from the inner domain τ. Unlike the conventional meaning of partition ESSENCE partitions do not necessarily cover all values of τ, they cover a subset of values. Consider the following partition domain with unknown but bounded number of parts.

```
find p : partition (maxNumParts n) from τ
```

This partition will be refined into a set of sets of τ, and additional constraints will be posted to maintain properties of a partition. Both levels of sets in the generated refinement domain introduce conditional symmetry, and these are broken by reusing the implementation for set domains.

3.5 Functions

In ESSENCE function domains are partial unless modified by the `total` attribute. Consider the following partial function domain, which has a bounded size.

```
find f: function (maxSize n) int(a..b) --> τ
```

The explicit representation of `f` is as follows.

```
find fVal: matrix indexed by [int(a..b)] of τ
find fUsed: matrix indexed by [int(a..b)] of bool
such that (sum i : int(a..b). fused[i]) <= n
```

This representation introduces conditional symmetry when items in `fVal` are unused, indicated by `fUsed` taking the value `false`. This conditional symmetry is broken using the following additional constraint.

```
forAll i : int(a..b) . !fUsed[i] -> dontCare(fVal[i])
```

4 Handling `dontCare` in CONJURE

This section presents the handling of `dontCare` constraints in CONJURE. We begin by defining the `dontCare` constraint and how it is implemented. We will then show how structural constraints and `dontCare` constraints are handled for nested domains.

The `dontCare` constraint takes as an argument a decision variable of any domain and forces it to take a unique assignment. The assignment must be unique but it does not need to maintain the invariants of the domain: care is taken to ensure that other structural constraints are not posted together with `dontCare` constraints as the two would conflict. The implementation of `dontCare` is straightforward: `dontCare` on a decision variable with an abstract domain is rewritten into a `dontCare` on the representation of the decision variable. For example a `dontCare` on a `partition` variable will be rewritten into a `dontCare` on the representation of it which has a `set of set` domain. Other abstract domains are handled similarly. `dontCare` constraints on matrix and tuple domains are rewritten into a conjunction of `dontCare`s on the elements of the domain. After successive application of such rewrites, the model only contains `dontCare` constraints on Boolean and integer domains. At this stage CONJURE rewrites the `dontCare` constraint into an unary equality constraint using the lowest value of the domain. The result is a valid ESSENCE′ model: **no** modification of the underlying constraint modelling and solving systems is required.

Refinement rules to select representations in CONJURE operate on domains and CONJURE applies them both when they are at the top level and when they are nested inside another domain constructor. For example, the domain `set of function A -> B` represents a set of functions mapping values from A to B. First, CONJURE chooses a representation for the outer set and refines it; then, the inner function is refined. During the refinement of the inner function, structural constraints need to be generated. These constraints need to be posted only to the active parts of the outer set, namely they need to be guarded using the switch variables. Conditionally applying structural constraints of the nested domains at the outer level is called *lifting*.

Figure 2 presents an example of conditional lifting of structural constraints. Figure 2a gives an ESSENCE problem specification which contains a variable size set which contains another abstract domain in it. Figure 2b gives the intermediate state, after refining the outer set and adding its structural constraints. Finally, Figure 2c gives the result of refining the nested domain nested inside a set domain. The structural constraints of the inner type are only posted on the active parts of the outer set.

The same technique is used for every representation in CONJURE that has active and inactive parts. Each representation only needs to report how to selectively post constraints to active parts of the decision variables used.

```
find x : set (maxSize 5) of A
```

(a) Input ESSENCE specification. A can be any domain.

```
find xused : matrix indexed by [int(1..5)] of bool
find xval  : matrix indexed by [int(1..5)] of A
such that
  forAll i: int(1..n-1) . xused[i+1] -> xused[i],
  forAll i : int(1..4) .
    xused[i] /\ xused[i+1] -> xval[i] .< xval[i+1],
  forAll i : int(1..5) .
    !xused[i] -> dontCare(xval[i])
```

(b) After the outer set is refined.

```
find xused : matrix indexed by [int(1..5)] of bool
find xval' : matrix indexed by [int(1..5)] of A'
such that
  forAll i: int(1..n-1) . xused[i+1] -> xused[i],
  forAll i : int(1..4) .
    xused[i] /\ xused[i+1] -> xval'[i] .< xval'[i+1],
  forAll i : int(1..5) .
    !xused[i] -> dontCare(xval'[i]),
  forAll i : int(1..5) .
    xused[i] -> structural(xval'[i])
```

(c) A refined. The structural constraint for A, which is imposed on the elements of `xval'`, is posted conditionally.

Figure 2: Lifting structural constraints conditionally.

5 Interaction with Search

It has been observed previously [15] that, due to bad interactions with the search strategy, adding symmetry breaking constraints can actually increase search effort. This is because the first solution that would have been found is removed by the symmetry breaking constraints. In practice, however, this is usually not a concern: the reduction in the size of the search space makes up for this effect, and the search required to find all solutions will always be smaller, given a static variable and value ordering. Furthermore, the symmetry breaking constraints themselves provide strong information as to how to organise the search to avoid conflicts.

Nonetheless, it is worth noting that exactly the same problem arises when breaking conditional symmetries using `dontCare`. Consider the set refinement given in Section 3.1. This refines a set `s` to two matrices `sVal` and `sUsed`. For the purposes of this example, we will set the parameters in this example to `n=3`, τ=`int(1..3)`. Consider search first assigning `sVal[3]` the value 2. The `dontCare` constraint implies that `sUsed[3]` is true, which further implies `sUsed[2]` and `sUsed[1]` are also true. This forces the set to be size 3. If instead there were no `dontCare` constraints, then we would still have to branch on `sUsed`. In particular, if the `dontCare` constraints were not present, search could have set each element of `sUsed` to false. If the only solution to our problem requires `s` $= \{\}$, this would find the solution faster.

However, as our experiments show, as with traditional symmetry breaking, benefits of effective conditional symmetry breaking greatly outweigh the possible small loss caused by a bad variable ordering.

6 Experiments

We ran two simple experiments to illustrate the effectiveness of automated conditional symmetry breaking in CONJURE by counting the number of solutions to ESSENCE problem specifications with and without `dontCare` constraints. The first also demonstrates that arbitrary combinations of nested types can be handled, even with conditional symmetries in each. In these experiments SAVILEROW and MINION were run with their default options on a 32-core AMD Opteron 6272 at 2.1 GHz.

First, we generated 25 ESSENCE specifications. Each contains a single decision variable with a 3-level nested domain, but no constraints. The innermost domain is always an integer domain, and we generate all combinations of 5 domain constructors in ESSENCE for the other layers. The outer two layers have a bounded size of 2, so can also be empty of or size 1, meaning that each layer will require additional `dontCare` constraints. Moreover, the structural constraints of the inner layer will need to be posted conditionally as described in Figure 2. CONJURE contains multiple refinement options for all of the domains in this experiment. In some cases it is able to generate thousands of models for one problem. However, since the conditional symmetry breaking constraints are needed in all of these models we only picked one model per problem using the Compact heuristic [1].

Table 1 presents the number of solutions for the same problem specification with and without conditional symmetry breaking constraints. The results are as expected: models with `dontCare` constraints have fewer solutions than those without. When finding all solutions for a model without `dontCare` constraints many of the generated solutions are symmetric to other solutions. The most extreme cases involve partitions, and can produce hundreds of millions of solutions when there are only ten symmetrically distinct ones. Using `dontCare` constraints, these symmetric solutions are avoided and the solver doesn't need to waste effort searching through them.

For the second experiment, we refer to the ESSENCE specification of the Dominating Queens problem given in Figure 1. The specification contains a partial function. We refined the specification for each $n \in \{4 \dots 14\}$, with and without `dontCares`. Figure 3 plots the total time taken by both SAVILEROW and MINION to translate and solve the problem instance. For all but the smallest instance, the model with `dontCares` is solved faster, for $n = 8$ more than 430 times faster. In this experiment a time limit of one hour was applied to MINION. SAVILEROW always took less than 8 seconds. Without `dontCares`, the solver timed out for $n \in \{9 \dots 14\}$, but with `dontCares` we found it scales considerably better, timing out for $n \in \{12 \dots 14\}$.

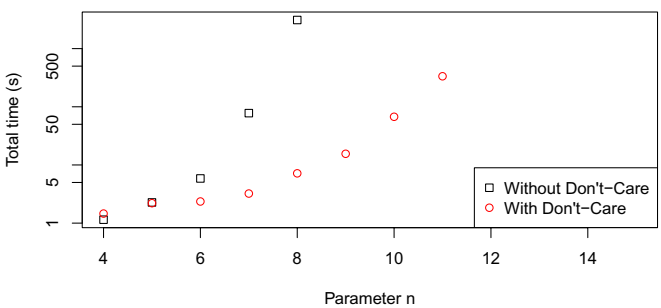


Figure 3: Plot of total time to solve Dominating Queens.

7 Consistent Symmetry Breaking

A well known issue when using constraints to break multiple sets of symmetries in the same problem is that the constraints can conflict, leading to lost solutions (see e.g. [9]). This problem does not occur when CONJURE breaks symmetries and conditional symmetries introduced during refinement. The reason for this is simple: each symmetry is broken as soon as it is introduced, allowing us to handle each introduced symmetry group in isolation.

To elaborate, one important feature of CONJURE is that during refinement we have a valid model after the application of each refinement rule (these partially-refined specifications include some constructs internal to CONJURE not in ESSENCE). Therefore when we introduce a conditional symmetry during refinement, and then immediately remove it by the addition of new constraints, at no point simultaneously are there two model symmetries that we have to break consistently. If, on the other hand, we delayed breaking symmetry until refinement was complete, we would then have to break all symmetries in a consistent manner.

The symmetry breaking constraints generated by CONJURE cannot conflict with any constraints provided by the user either. CONJURE only breaks the symmetry introduced by itself. For this purpose, it posts symmetry breaking constraints on the concrete decision variables it generates, the users do not have access to these variables and they cannot write any conflicting constraints in terms of them.

Using the refinement rules in this paper, refining any ESSENCE specification with a single variable with CONJURE produces a model with an identical number of solutions. This implies we have broken all symmetries which would lead to one ESSENCE solution being duplicated as multiple ESSENCE′ solutions. We only need to ensure each refinement rule in isolation achieves this goal, then the application of all rules will achieve this.

We have focused in this paper on model symmetry. While the abstraction of the ESSENCE language naturally lends itself to writing ESSENCE specifications without symmetry, we do expect that some ESSENCE specifications will contain symmetries and conditional symmetries. Assuming this symmetry has been detected (a topic not addressed in this paper) and broken consistently by adding additional constraints to the specification prior to refinement (for example via the Crawford Ordering [8]) there will be no consistency issue with the way in which CONJURE breaks model symmetry.

8 Other uses of `dontCare` in refinement

The `dontCare` operator has other uses beyond type refinement. For example [14] discusses how to deal with undefined values (for example dividing an integer value by 0) during refinement.

Consider the refinement of `(x/y=z) <-> B`, for integer variables `x`,`y`, `z` and Boolean `B`. In MiniZinc 1.6, this produces the following refinement (rewritten as ESSENCE):

```
find b1,b2,B: bool
find i1,i2,x,y,z: int(0..3)

such that
    (b1 /\ b2) = B,     x / i1 = i2,
    (z = i2) <-> b1,    (y = i1) <-> b2,
    (y != 0) <-> b2
```

We want to ensure that for every assignment to `x`, `y`, `z` and `B` which satisfy `(x/y=z) <-> B`, there is exactly one assignment to the auxiliary variables `b1`, `b2`, `i1` and `i2` which satisfies all the constraints. When `y != 0`, this is the case. On the other hand, when `y = 0` then `i1` and `i2` can be assigned any value under the conditions that `i1 != 0` and `x/i1=i2`. We will show how to remove this

Outer \ Inner	set		mset		function		relation		partition	
dontCare	With	Without	With	Without	With	Without	With	Without	With	Without
set	11	38	22	87	46	632	67	297	15	845
mset	19	58	34	129	73	928	101	441	25	1315
function	25	64	49	144	100	1024	144	484	36	1444
relation	137	632	667	3222	4042	174512	7382	36542	296	318452
partition	41	310978	352	9092502	10	$\geq$ 277220736	88574	$\geq$ 198611820	208	$\geq$ 138135600

Table 1: Number of solutions with and without `dontCare` constraints. A $\geq$ indicates number of solutions found within 1 hour CPU timeout.

conditional symmetry. We must first remove 0 from the domain of `i1`. This does not alter the set of solutions, as `y = 0` implies `y != i1` and `y != 0` implies `y = i1`. After removing 0 from the domain of `i1`, we can add the constraint `¬b2 -> dontCare(i1)`. This eliminates all conditional symmetry by ensuring `i1` only takes a single value when `y != 0`, which further implies a single valid assignment for `i2` by the constraint `x/i1=i2` and for `b1` by the constraint `(z = i2) <-> b1`.

9 Conclusion

We have presented a systematic method by which the automated constraint modelling tool CONJURE can break conditional symmetry as it enters a model during refinement. Our method extends, and is compatible with, our previous work on automated symmetry breaking in CONJURE. Excepting unnamed types, which are a technical part of ESSENCE designed to encapsulate a particular part of symmetry, the result is the complete and automatic removal of model symmetry for the entire problem class represented by the output model - a significant step forward for automated constraint modelling.

Acknowledgements This work was supported by UK EPSRC EP/K015745/1. Jefferson is supported by a Royal Society University Research Fellowship.

REFERENCES

[1] Ozgur Akgun, Alan M Frisch, Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale, 'Automated Symmetry Breaking and Model Selection in Conjure', in *Principles and Practice of Constraint Programming - CP 2013*, (2013).

[2] Ozgur Akgun, Ian Miguel, Christopher Jefferson, Alan M. Frisch, and Brahim Hnich, 'Extensible automated constraint modelling', in *AAAI-11: Twenty-Fifth Conference on Artificial Intelligence*, (2011).

[3] Nicolas Beldiceanu and Helmut Simonis, 'A model seeker: Extracting global constraint models from positive examples', in *18th International Conference on Principles and Practice of Constraint Programming*, pp. 141–157, (2012).

[4] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan, 'Leveraging the learning power of examples in automated constraint acquisition', in *10th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pp. 123–137. Springer Berlin Heidelberg, (2004).

[5] Christian Bessiere, Remi Coletta, Frederic Koriche, and Barry O'Sullivan, 'Acquiring constraint networks using a SAT-based version space algorithm', in *AAAI 2006*, pp. 1565–1568, (2006).

[6] John Charnley, Simon Colton, and Ian Miguel, 'Automatic generation of implied constraints', in *Proc. of ECAI 2006*, pp. 73–77, (2006).

[7] Remi Coletta, Christian Bessiere, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joel Quinqueton, 'Semi-automatic modeling by constraint acquisition', in *9th International Conference on Principles and Practice of Constraint Programming*, pp. 812–816, (2003).

[8] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy, 'Symmetry-breaking predicates for search problems', *KR*, **96**, (1996).

[9] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh, 'Breaking row and column symmetries in matrix models', in *Proceedings CP 2002*, pp. 462–476.

[10] Pierre Flener, Justin Pearson, and Magnus Ågren, 'Introducing ESRA, a relational language for modelling combinatorial problems', in *LOPSTR 2003*, pp. 214–232, (2003).

[11] A. M. Frisch, C. Jefferson, B. Martinez Hernandez, and I. Miguel, 'The rules of constraint modelling', in *Proc. of the IJCAI 2005*, (2005).

[12] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel, 'Essence: A constraint language for specifying combinatorial problems', *Constraints 13(3)*, 268–306, (2008).

[13] Alan M. Frisch, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel, 'Symmetry in the generation of constraint models', in *Proceedings of the International Symmetry Conference*, (2007).

[14] Alan M Frisch and Peter J Stuckey, 'The proper treatment of undefinedness in constraint languages', in *Principles and Practice of Constraint Programming-CP 2009*, 367–382, Springer, (2009).

[15] Ian P. Gent, Warwick Harvey, and Tom Kelsey, 'Groups and constraints: Symmetry breaking during search', in *CP*, ed., Pascal Van Hentenryck, volume 2470 of *Lecture Notes in Computer Science*, pp. 415–430. Springer, (2002).

[16] Ian P. Gent, Tom Kelsey, Steve Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith, 'Conditional symmetry breaking', in *CP*, ed., Peter van Beek, volume 3709 of *Lecture Notes in Computer Science*, pp. 256–270. Springer, (2005).

[17] Ian P. Gent, Karen Petrie, and Jean-Francois Puget, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapter Symmetry in Constraint Programming, 329–376, Elsevier Science Inc., New York, NY, USA, 2006.

[18] PB Gibbons and JA Webb, 'Some new results for the queens domination problem', *Australasian Journal of Combinatorics*, **15**, (1997).

[19] Brahim Hnich, 'Thesis: Function variables for constraint programming', *AI Commun*, **16**(2), 131–132, (2003).

[20] Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey, 'Data independent type reduction for zinc', in *ModRef10*, (2010).

[21] A. Lallouet, M. Lopez, L. Martin, and C. Vrain, 'On learning constraint problems', in *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pp. 45–52, (2010).

[22] James Little, Cormac Gebruers, Derek G. Bridge, and Eugene C. Freuder, 'Using case-based reasoning to write constraint programs', in *CP*, p. 983, (2003).

[23] Toni Mancini and Marco Cadoli, 'Detecting and breaking symmetries by reasoning on problem specifications', in *Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, pp. 165–181. Springer Berlin Heidelberg, (2005).

[24] 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 13(3)*, (2008).

[25] Christopher Mears, Todd Niven, Marcel Jackson, and Mark Wallace, 'Proving symmetries by model transformation', in *17th International Conference on Principles and Practice of Constraint Programming*, CP'11, pp. 591–605, Berlin, Heidelberg, (2011). Springer-Verlag.

[26] P. Mills, E.P.K. Tsang, R. Williams, J. Ford, and J. Borrett, 'EaCL 1.5: An easy abstract constraint optimisation programming language', Technical report, University of Essex, Colchester, UK, (December 1999).

[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack., 'Minizinc: Towards a standard CP modelling language', in *Proc. of CP 2007*, pp. 529–543, (2007).

[28] Andrea Rendl, *Thesis: Effective Compilation of Constraint Models*, Ph.D. dissertation, University of St. Andrews, 2010.

[29] Pascal Van Hentenryck, *The OPL Optimization Programming Language*, MIT Press, Cambridge, MA, USA, 1999.