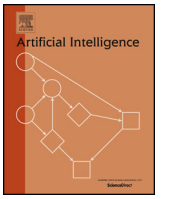
Artificial Intelligence 310 (2022) 103751

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



CONJURE: Automatic Generation of Constraint Models from Problem Specifications

Özgür Akgün [a,*], Alan M. Frisch [b], Ian P. Gent [a], Christopher Jefferson [a], Ian Miguel [a], Peter Nightingale [b]

[a] School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK
[b] Department of Computer Science, University of York, Deramore Lane, Heslington, York YO10 5GH, UK

ARTICLE INFO

Article history:
Received 22 November 2021
Received in revised form 23 May 2022
Accepted 6 June 2022
Available online 9 June 2022

Keywords:
Constraint modelling
Constraint programming
Combinatorial optimization
Constraint satisfaction problem

ABSTRACT

When solving a combinatorial problem, the formulation or *model* of the problem is critical to the efficiency of the solver. Automating the modelling process has long been of interest because of the expertise and time required to produce an effective model of a given problem. We describe a method to automatically produce constraint models from a problem specification written in the abstract constraint specification language ESSENCE. Our approach is to incrementally *refine* the specification into a concrete model by applying a chosen *refinement rule* at each step. Any non-trivial specification may be refined in multiple ways, creating a space of models to choose from.

The handling of symmetries is a particularly important aspect of automated modelling. Many combinatorial optimisation 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: modelling symmetries. We show how modelling symmetries may be broken automatically as they enter a model during refinement, obviating the need for an expensive symmetry detection step following model formulation.

Our approach is implemented in a system called CONJURE. We compare the models produced by CONJURE to constraint models from the literature that are known to be effective. Our empirical results confirm that CONJURE can reproduce successfully the kernels of the constraint models of 42 benchmark problems found in the literature.

1. Introduction

Efficient decision-making is of central importance to a modern society. It is natural to represent and reason about decision-making problems in terms of constraints. For example, in scheduling a football league many constraints occur, such as: every team has to play every other, home and away; every match must be assigned a set of officials, and no official or team can be in two places at once; no team should be scheduled to play more than, say, four consecutive away games. Constraint programming [1] offers a means by which solutions to such problems can be found automatically. Constraint

* Corresponding author.
E-mail addresses: ozgur.akgun@st-andrews.ac.uk (Ö. Akgün), alan.frisch@york.ac.uk (A.M. Frisch), ian.gent@st-andrews.ac.uk (I.P. Gent), caj21@st-andrews.ac.uk (C. Jefferson), ijm@st-andrews.ac.uk (I. Miguel), peter.nightingale@york.ac.uk (P. Nightingale).

https://doi.org/10.1016/j.artint.2022.103751

```
language Essence 1.3
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched : set (size w) of
             partition (regular, numParts g, partSize s)
             from Golfers
such that
  forAll g1, g2 : Golfers, g1 < g2 .
    (sum week in sched . toInt(together({g1, g2}, week))) <= 1
```

Fig. 1. An Essence problem specification of the Social Golfers Problem (Problem 10 at CSPLib.org). In a golf club there are a number of golfers who wish to play together in g groups of size s. Find a schedule of play for w weeks such that no pair of golfers play together more than once.

solving of a given problem proceeds in two phases. First, the problem is *modelled* as a set of *decision variables*, and a set of *constraints* on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. The *domain* of potential values associated with each decision variable corresponds to the options for that choice. In our football league example, one might have two decision variables per match to represent each of the home and away teams. The second phase consists of using a constraint solver to find solutions to the model: assignments of values to decision variables satisfying all constraints.

There are typically many possible models for a given problem, and the model chosen can dramatically affect the efficiency of constraint solving. This presents a serious obstacle for non-expert users, who have difficulty in formulating a good (or even correct) model from among the many possible alternatives. Modelling is therefore a critical bottleneck in the process of constraint solving, considered to be one of the key challenges facing the constraints field [2].

It is desirable, therefore, to automate constraint modelling as far as possible. Several approaches have been taken to automate aspects of constraint modelling. Some approaches learn models from, variously, natural language [3], positive or negative examples [4–6], membership queries, equivalence queries, partial queries [7,8], generalisation queries [9] or arguments [10]. Other approaches include: automated transformation of medium-level solver-independent constraint models [11–17]; deriving implied constraints from a constraint model [18–22]; case-based reasoning [23]; and refinement of abstract constraint specifications [24] in languages such as ESRA [25], Essence [26], $\mathcal{F}$ [27] or Zinc [28–30]. We focus herein on the refinement approach, where a user writes a constraint specification describing a problem above the level of abstraction at which modelling decisions are made. In Section 8 we discuss in more detail alternative approaches to automated constraint modelling by this method.

This paper presents the automated constraint modelling system Conjure, which serves to demonstrate the efficacy of the refinement-based approach. A problem is input to Conjure in Essence, an abstract constraint specification language. Essence's support for 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 allows a problem to be specified *without* committing to constraint modelling decisions. To illustrate, consider the fragment of the Essence specification of the Social Golfers Problem [31] presented in Fig. 1. Given a number of weeks (w), a number of groups (g) and a group size (s), the problem is to find a schedule of play over the w weeks for the $g \times s$ golfers divided into g groups of size s, subject to a socialisation constraint among the golfers that stipulates that no pair of golfers play together more than once. The Social Golfers Problem is naturally conceived as finding a set of partitions of golfers subject to some constraints, which can be specified in Essence via a *single* abstract decision variable, as presented in the figure where the variable is `sched`.

Since these abstract types are not supported directly by constraint solvers,[1] an Essence specification must be transformed (refined) into a constraint model. Automating this process presents a considerable challenge and the contributions of this work are in meeting that challenge. Principal among these is a carefully designed rule-based architecture implemented in Conjure to refine an Essence specification into a constraint model. One key contribution is that Conjure can refine nested types without resorting to enumerating the values of the inner type (for example, refining a set of sets of integers without enumerating all possible values of the inner set). This capability is vital to refining many of the Essence specifications that we use in the evaluation. As we will demonstrate, different rule application pathways produce different constraint models, supporting an automated model selection process among the many possible alternatives. This approach also facilitates the automated production of *channelled* constraint models [32], in which a single abstract decision variable is refined in multiple ways. Channelling constraints are elegantly generated for an abstract decision variable A by creating the equality $A = A$ and refining it with two different representations of A, thus ensuring the two representations take the same abstract value in all solutions. Channelled models have previously been created manually by experts, typically in an effort to simplify the statement of the problem constraints so as to strengthen the inference of the constraint solver and reduce search.

A further important contribution of our rule-based architecture is in the treatment of *symmetry*, a structure-preserving transformation. In the context of a constraint problem, given a solution to a problem instance we can obtain another symmetric solution. Symmetry can lead to redundant search: if the constraint solver reaches a dead end in its search for a

[1] Set variables are a notable exception, which are widely supported. However, the solvers that support set variables do not offer a choice as to the underlying representation of the set, and do not support nested sets.

solution, 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, which we have called *modelling* symmetries [33–35]. Modelling symmetries occur naturally as abstract decision variables are refined into constraint models.

As a simple example consider representing a set of size n by a vector of n variables, constrained to take distinct values. 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 to the model the constraint that the integers appear in the vector in increasing order. However, this simple approach cannot be used directly if the elements of the set are themselves (for example) sets of multisets. As we will discuss, our rule-based architecture can recognise when modelling symmetries arise in the refinement of a constraint model and add symmetry breaking constraints to deal with complex symmetries of this type. This obviates the need for an expensive symmetry detection step following model formulation, as used by other approaches [36–38]. When a refinement performed by Conjure introduces symmetry, the symmetry is broken consistently and completely by the addition of symmetry-breaking constraints. In several cases this also allows for improved refinement of Essence expressions. 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.

Our final contribution is an empirical evaluation of the coverage of the model space provided by Conjure. In an extensive set of experiments we show that, for a wide variety of problems, the substantial majority of models crafted manually by human experts can be automatically generated by Conjure from Essence specifications of those problems. In addition, we present a simple and lightweight heuristic for choosing among the models generated by Conjure. The CompactEP heuristic is often able to select a good model for a given problem specification. We evaluate the accuracy of CompactEP on a wide range of problems. Rather than focusing on the runtime performance of models with particular solvers and instance sets (which would give a very limited picture of model quality), we performed a qualitative comparison of the generated models with previously published models in Section 7.

Our approach to the refinement of types and expressions is from the outside-in, which allows refinement rules to handle a single layer of a type, or single operator, at a time – although multiple types or operators can be handled where this can improve the refinement. Quantified expressions are handled generically in a way that is independent of which quantifier is used, by separating the gathering of values to be quantified over and application of the quantifying operator. Conjure is designed to be extended with further types, attributes and operators in the future – several types, including sequences, have been added to Essence since the first release of Conjure.

The work presented in this paper summarises and extends over fifteen years of our work on automated constraint modelling. Our earliest work on refinement-based automated constraint modelling appeared between 2002 and 2005 [39–42,24]. We introduced the Essence language in 2005 [43,44], which is the subject of a separate journal article [26]. Following the presentation of initial prototypes [24,45] the first full version of Conjure was presented in 2011 [46] then extended to handle automated symmetry breaking [34,35], and presented in detail in Akgun's thesis [47]. Herein, we give a complete overview of Conjure, including the most recent advances.

1.1. Contributions

In summary, our main contributions are as follows:

- Conjure is unique in refining problem class specifications to class-level constraint models.
- Multiple models are generated from one Essence specification by following different rule application pathways.
- Conjure is able to refine nested abstract types (for example, a set of sets of integers) without enumerating all possible values of the inner type (in this example, set of integers).
- Symmetry introduced during refinement is broken consistently and completely.
- Conjure is able to generate channelled models by representing an abstract decision variable in more than one way, with an elegant mechanism for producing channelling constraints from a simple equality constraint.
- Model selection is achieved via the simple and lightweight CompactEP heuristic, which is shown to select good models in many cases.
- The system is evaluated comprehensively on 42 problem classes from CSPLib [48], demonstrating that Conjure is able to generate models similar to models in the literature produced by experts.

2. Conjure by Example

This section illustrates the operation of Conjure on a simple problem specification. It exemplifies some constructs of the input language Essence and the output language Essence Prime. There are a large number of *refinements* that are applied to transform a full Essence specification into a concrete Essence Prime constraint model. The goal of this example is to highlight the most important kinds of refinements before we describe them in their full generality. We include forward references to later sections where appropriate.

```
1  language Essence 1.3
2  given object new type enum
3  given weight, value : function (total) object --> int(1..)
4  given maxWeight: int(1..)
5  find knapsack : set of object
6  maximising sum i in knapsack . value(i)
7  such that (sum i in knapsack . weight(i)) <= maxWeight
```

Fig. 2. An Essence specification of the Knapsack Problem.

2.1. The Knapsack Problem in Essence

Fig. 2 shows an Essence specification for the Knapsack Problem. We have chosen this familiar problem to illustrate the basics of refinement. The Knapsack specification does lack some of the more sophisticated features of Essence, such as nested types, and we will explain how Conjure treats these in later sections.

Lines 2–3 specify the problem class parameters: an enumerated type of objects; a weight and a value per item, represented as total functions; and the maximum weight of the knapsack. Line 5 specifies the single decision variable, the set of objects to be placed in the knapsack. Line 6 specifies the objective function, which is to maximise the value of the collection of items in the knapsack. Finally, line 7 specifies the capacity constraint.

Some features of Essence, such as the function domains in this specification, are not supported by conventional constraint modelling languages. Therefore, they need to be refined to use supported features like integer and matrix domains. Moreover, the problem constraints are stated in terms of Essence domains, which also need to be refined accordingly.

Some refinement steps are simple, such as replacing enumerated domains with isomorphic integer domains. Others are more complex, such as choosing a representation for abstract decision variables and refining abstract constraint expressions. In the rest of this section we focus on the set decision variable `knapsack` and present multiple ways of refining it.

2.2. Choosing Representations

Before applying any modelling refinements, Conjure traverses the entire model and labels every reference to abstract decision variables (Section 3.1) with a representation decision (Section 4.1). In this example, the `knapsack` variable is an abstract decision variable and is referenced in two places, on lines 6 and 7.

We consider two representations for a set domain in this section: the *Explicit* representation and the *Occurrence* representation. The *Explicit* representation uses a matrix of decision variables, representing the elements of the set, together with a single integer variable, representing the cardinality of the set. Structural constraints (Section 4.1) are posted to ensure these variables represent valid set values. The *Occurrence* representation uses a Boolean matrix of decision variables, indexed by the domain of possible elements of the set. In this representation, a `true` value at a certain index of the matrix indicates set membership.

Conjure may use either of these representations for each reference to the `knapsack` variable.[2] Choosing multiple representations for the same abstract decision variable leads to channelled models (Section 4.3).

2.3. The Explicit representation

Choosing the *Explicit* representation for both references to the `knapsack` variable leads to the addition of new variable declarations and structural constraints to the model, as shown in Fig. 3. The matrix `knapsack_ Explicit` represents the elements of the set, and the integer variable `knapsack_Size` represents the cardinality of the set. The first structural constraint both enforces distinctness and achieves symmetry breaking by sorting the entries in the matrix. The sorting is only enforced up to the cardinality of the set, since entries after this point are not members of the set. The second structural constraint assigns the variables after the `knapsack_Size` marker to take an arbitrarily chosen value of their domain, as described in Section 4.1.3.

The two references to `knapsack` are refined using the *Explicit* representation. Fig. 4 shows the refinement of only one of the expressions, the other expression is refined similarly. The `sum` expression quantifying over the set decision variable is refined to another `sum` expression quantifying over a simple integer domain. We use a multiplication with the set membership condition inside the quantified expression. This allows us to exclude entries in the matrix that do not represent members of the set.

2.4. The Occurrence representation

Similarly, choosing the *Occurrence* representation for both references to the `knapsack` variable leads to the addition of a new variable declaration to the model. This is shown in Fig. 5. The matrix `knapsack_Occurrence` represents the set.

[2] These two representations are given as examples here; there are more representation options in Conjure.

```
find knapsack_Explicit: matrix indexed by
                          [int(1..|object|)] of object
find knapsack_Size    : int(0..|object|)
such that
  forAll i : int(1..|object|) .
    i + 1 <= knapsack_Size ->
      knapsack_Explicit[i] < knapsack_Explicit[i+1],
  forAll i : int(1..|object|) .
    i > knapsack_Size ->
      dontCare(knapsack_Explicit[i])
```

Fig. 3. The new declarations and structural constraints after choosing the *Explicit* representation.

```
(sum i : int(1..|object|) .
    toInt(i <= knapsack_Size) * weight(knapsack_Explicit[i]))
        <= maxWeight
```

Fig. 4. Expression refinement after choosing the *Explicit* representation.

```
find knapsack_Occurrence : matrix indexed by [object] of bool
```

Fig. 5. The new declaration after choosing the *Occurrence* representation.

```
(sum i : object . toInt(knapsack_Occurrence[i]) * weight(i)) <= maxWeight
```

Fig. 6. Expression refinement after choosing the *Occurrence* representation.

```
forAll i : object . knapsack_Occurrence[i] ->
  exists j : int(1..|object|) .
    j <= knapsack_Size /\ knapsack_Explicit[j] = i,
forAll i : int(1..|object|) . i <= knapsack_Size ->
  knapsack_Occurrence[knapsack_Explicit[i]]
```

Fig. 7. Channelling constraints between the *Explicit* and *Occurrence* representations.

A `true` assignment at index `i` of the matrix indicates that value `i` is in the set. This representation does not introduce any symmetry, and it does not require any structural constraints to be posted. This is because every assignment to the Boolean matrix corresponds to a unique assignment to the original set variable.

The two references to the `knapsack` variable are refined using the *Occurrence* representation. Fig. 6 shows the refinement of one of the expressions, and the other is refined similarly. The `sum` expression quantifying over the set decision variable is refined to another `sum` expression quantifying over the potential members of the set. Once again we use multiplication to exclude the values that are not members of the set.

2.5. Channelled models

Suppose we chose more than one representation for a single abstract decision variable. Each representation would be generated as above, but they would also need to be connected together to ensure the representations all represent the same value of the original decision variable. Models with more than one representation are called *channelled* models, and the constraints connecting the representations are *channelling constraints*.

In a channelled model with two representations, both sets of decision variables and structural constraints are added to the model. Each reference to the decision variable is refined using its chosen representation. The channelling constraints are generated by posting an equality constraint (in this example, `knapsack=knapsack`) and tagging the two occurrences of the decision variable with different representations. The equality is then refined using the same refinement procedures that are applied to any constraint. For our running example, the channelling constraints are given in Fig. 7. The first constraint ensures all members of the *Occurrence* representation are also members of the *Explicit* representation, and the second constraint ensures the same holds in the opposite direction.

Table 1
Domains and domain constructors (parameterised domains) in Essence. Arguments of domain constructors are denoted τ or τ_1, τ_2, etc. Domains and domain constructors may be nested arbitrarily.

Domain	Handling
Concrete domains (Atomic)	
`bool`	Kept unchanged
`int`	Kept unchanged
enumerated	Mapped to integers
unnamed	Mapped to integers
Concrete domains (Compound)	
`tuple (`$\tau_1, \tau_2, \ldots$`)`	Separated into components
`record {(alice,` τ_1`), (bob,` τ_2`), ...}`	Separated into components
`variant {(alice,` τ_1`), (bob,` τ_2`), ...}`	Separated into components
`matrix [`τ_1`,` τ_2`, ...,` τ_n`] of` τ	Kept unchanged
Abstract domains	
`set of` τ	Refined
`mset of` τ	Refined
`sequence of` τ	Refined
`function` τ_1 `->` τ_2	Refined
`relation of (`$\tau_1, \tau_2, \ldots$`)`	Refined
`partition from` τ	Refined

2.6. Summary

In this section we have illustrated how Conjure generates multiple diverse models from a single specification by choosing representations of the abstract decision variables. In the following sections we describe the Conjure system, its input language Essence, and the set of refinement rules and representations that allow us to generate a diverse set of models. Section 3.3 presents Conjure in the context of a pipeline of tools and languages.

3. Automated Modelling in Conjure

In this section we set the scene for automated modelling by describing Conjure itself, the toolchain it sits within, and the languages produced and consumed by Conjure and the other tools. First we summarise the Essence language consumed by Conjure and highlight its most important features. We then summarise the Essence Prime language produced by Conjure, and the tool Savile Row that translates Essence Prime to the language of a target solver.

3.1. Summary of the Essence Language

This section provides a summary of the current state of the Essence language sufficient to describe the operation of Conjure. For further details the reader is referred to the original journal paper describing Essence [26] and the frequently updated documentation accompanying the Conjure release [49].

Conjure takes as input an abstract problem specification written in Essence and automatically generates Essence Prime models as output. Essence is a high-level problem specification language providing a rich set of built-in domains and domain constructors (parameterised domains), such as multi-sets, functions, and partitions. Decision variables can have these domains so as to precisely encode what they *mean*, and to avoid the need to *model* these complex domains via multiple decision variables with simpler domains. Essence domains that are not directly represented in Essence Prime are called *abstract* domains and domains that are shared between the two languages are called *concrete* domains (Boolean, int, and matrices of these). We also characterise domains as *compound* when they contain multiple elements (such as a tuple or matrix). Tuples and records contain a fixed number of fields. Fields in a tuple domain are identified by their position and fields in a record domain are identified by the field name. Variants are tagged unions: they contain a single value for one of the components, tagged by the name of the component. The full set of domains and domain constructors in Essence and the handling of abstract and concrete domains is given in Table 1. Domains and domain constructors may be nested arbitrarily, allowing for rich domains such as a partition of sets of integers.

Unnamed types [26] may be unfamiliar so we briefly describe them here. An unnamed type represents a set of objects that are indistinguishable, such as the golfers of the Social Golfers Problem (Fig. 1). The elements of an unnamed type are not named or numbered individually, and so cannot be referred to directly in the specification. Unnamed types exist to provide an abstraction for sets of indistinguishable objects, allowing such sets to be specified without introducing symmetry. However, the current implementation of unnamed types in Conjure (mapping to integers) introduces symmetry. An implementation that does not do so is challenging and an important area of future work, as described in Section 4.1.2.

Domains are further specified by adding *attributes*, and each domain constructor has its own set of attributes that may be used with it. Attributes further restrict (i.e. make precise) an abstract domain, so the user of Essence does not need to use constraints to achieve the desired effect. For instance, a set variable may have a `minSize` attribute attached to it, which

Table 2
All domain attributes in Essence.

Domain	Attributes
`set of` τ	`size`, `minSize`, `maxSize`
`mset of` τ	`size`, `minSize`, `maxSize`, `minOccur`, `maxOccur`
`sequence of` τ	`size`, `minSize`, `maxSize`, `injective`, `surjective`, `bijective`
`function` τ_1 `->` τ_2	`size`, `minSize`, `maxSize`, `injective`, `surjective`, `bijective`, `total`
`relation of` $(\tau_1, \tau_2, \ldots)$	`size`, `minSize`, `maxSize`. For binary relations only: `reflexive`, `irreflexive`, `coreflexive`, `symmetric`, `antiSymmetric`, `aSymmetric`, `transitive`, `total`, `connex`, `Euclidean`, `serial`, `equivalence`, `partialOrder`
`partition from` τ	`numParts`, `minNumParts`, `maxNumParts`, `partSize`, `minPartSize`, `maxPartSize`, `regular`

Table 3
Operators of abstract types and matrices in Essence. In addition, equality, disequality, and ordering operators are provided for all types, and many types may be used as generators of comprehensions and quantifiers as shown in Table 4. For a full list of operators on all types and full definitions see the Conjure documentation [49].

Domain	Operators
`matrix` $[\tau_1, \tau_2, \ldots]$ `of` τ_n	`[x]` *(indexing)*, `[..]` *(slicing)*, *lexicographic ordering*, `sum`, `product`, `and`, `or`, `xor`, `min`, `max`
`set of` τ	`in`, `subset`, `subsetEq`, `supset`, `supsetEq`, `intersect`, `union`, `powerSet`, `-` *(difference)*, $\lvert x \rvert$ *(cardinality)*, `sum`, `product`, `and`, `or`, `xor`, `min`, `max`
`mset of` τ	`union`, `intersect`, `-` *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\lvert x \rvert$ *(cardinality)*, `in`, `freq`, `hist`, `sum`, `product`, `and`, `or`, `xor`, `min`, `max`
`sequence of` τ	`subsequence`, `substring`, $\lvert x \rvert$ *(cardinality)*, `defined`, `range`, `image`, `preImage`
`function` τ_1 `->` τ_2	`union`, `intersect`, `-` *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\lvert x \rvert$ *(cardinality)*, `defined`, `range`, `inverse`, `image`, `preImage`
`relation of` $(\tau_1, \ldots)$	`union`, `intersect`, `-` *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\lvert x \rvert$ *(cardinality)*, *relation application*, *relation projection*
`partition from` τ	$\lvert x \rvert$ *(cardinality)*, `together`, `apart`, `participants`, `parts`, `party`

ensures that the values of the decision variable are sets containing at least the given number of elements. The attributes of each domain constructor are given in Table 2.

Essence is statically typed and Conjure completely type-checks a specification before refining it. Each decision variable or parameter has a domain, and to obtain the corresponding type Conjure strips the attributes from the domain, replaces all `int(...)` with the type `int`, and replaces all subsets of enumerated types with the corresponding full enumerated type. Essence also has a rich collection of operators that allow concise expressions to be written on abstract types. For example, for functions there is an `inverse` operator, which ensures two functions are inverses of each other. For relations, relation projection lets us create a relation of smaller arity while fixing some of the components to a specific value. Excepting integer and Boolean operators, which may be found in the manual, the complete set of operators in Essence is summarised in Table 3, organised by the types to which they may be applied. Operators may be nested in any way that respects type-correctness.

Essence also provides *quantifiers* and *comprehensions* to construct complex expressions that are difficult or impossible to express using only the operators in Table 3. Quantifiers and comprehensions introduce local variables that take values from a domain or an abstract decision variable. For example, the knapsack specification in Section 2 contains the following sum quantifier, where `knapsack` is a decision variable of type `set of int`, and `value` is a function from objects to their monetary value. The quantifier calculates the total value of objects in the knapsack.

```
sum i in knapsack . value(i)
```

A quantifier has a keyword (`forAll`, `exists`, or `sum`), the quantified variable, a domain or abstract decision variable that defines the set of values that the quantified variable will take, and finally an inner expression (of type `int` for `sum` quantifiers, otherwise `bool`). A quantifier can be evaluated by binding the quantified variable to each value in turn, evaluating the inner expression for each value, then aggregating the results by conjunction, disjunction, or addition for the quantifiers `forAll`, `exists`, or `sum` respectively. Table 4 summarises the types of expressions that may be used to generate the set of values that the quantified variable will take, and the corresponding type of the quantified variable in each case.

Medium-level constraint modelling languages (such as Essence Prime and OPL [13]) typically have the quantifiers `forAll`, `exists`, and `sum` (and in some cases others such as `product`, `min`, `max`), but the quantified variable has type `int`, and the values are drawn from a domain with type `set of int`, not from an abstract domain or abstract decision variable. Quantifiers in Essence are substantially more general than those in Essence Prime, which does not have the abstract types.

Comprehensions in Essence create a one-dimensional matrix (a list). The list may then be aggregated to a single value using a function such as `and`, `or`, `xor`, `sum`, `product`, `min`, `max`, or global constraints like `allDifferent`. Lists generated via comprehensions can be used as arguments to several operators, in contrast quantified expressions are limited to `forAll`, `exists`, and `sum`. In common with quantified expressions, comprehensions have an inner expression and they introduce local variables whose values are drawn from an abstract domain or abstract decision variable. Comprehensions

Table 4
Types of expressions that can act as generators for quantified expressions and comprehensions. The type of the quantified variables changes depending on the type of the collection being quantified on.

Type of collection	Type of quantified variable	Quantified variable represents
`matrix [...] of` τ	τ	Member
`set of` τ	τ	Member
`mset of` τ	τ	Member
`sequence of` τ	`tuple(`*index*, τ`)`	Member and its sequence index
`function` τ_1 `->` τ_2	`tuple(`τ_1, τ_2`)`	Mapping in the function
`relation of (`τ_1,...`)`	`tuple(`τ_1,...`)`	Member of relation
`partition from` τ	`set of` τ	Part in the partition

also have *conditions*: Boolean expressions that act as a filter. The condition can contain references to decision variables, which is not possible in the comprehensions found in Essence Prime for example. Comprehensions (with aggregation functions) are more expressive than quantifiers, and they are used internally throughout Conjure in preference to quantifiers. The example above can be expressed as a comprehension as follows:

```
sum([ value(i) | i <- knapsack ])
```

As a final example of both quantifiers and comprehensions, suppose we wished to find a multiset of integers where all elements above 10 are even numbers. In the following Essence specification, the constraint on the elements of the multiset `M` is expressed using a quantifier and an implication.

```
find M : mset (maxSize 5) of int(1..20)
such that forAll i in M . i > 10 -> i % 2 = 0
```

The same constraint can also be expressed using a comprehension with a condition, as follows.

```
such that and([ i % 2 = 0 | i <- M, i > 10 ])
```

In both quantified expressions and comprehensions, all collection types can be used as generators. The type of the quantified variable is chosen based on the type of the generator.

3.2. Summary of the Essence Prime Language

Essence Prime [50] is a medium-level solver-independent constraint modelling language with some similarities to other modelling languages such as OPL [13] and MiniZinc [12]. Essence Prime was originally conceived as a subset of Essence without the abstract types. For the purposes of this paper, Essence Prime can be considered as a subset of Essence with the following restrictions:

1. There are no *abstract types* (sets, multisets, sequences, functions, relations, or partitions). Essence Prime supports decision variables and problem class parameters of type `int`, `bool`, and matrix of `int` and `bool`. Matrices may have any number of dimensions, and may be indexed by any integer domain.
2. Generators and conditions within comprehensions and quantifiers are not allowed to contain decision variables.

3.3. The Pipeline

Our modelling and solving pipeline is illustrated in Fig. 8. An Essence problem specification is given to Conjure, which refines the specification into a set of concrete models in Essence Prime. Both the specification and the model typically relate to a problem class, i.e. they both have problem class parameters that need to be instantiated before instances of the class can be solved. Conjure separately translates problem class parameters expressed in Essence into Essence Prime using the representations selected when refining the problem specification. This allows the user to solve multiple instances of the same problem class while only performing refinement once.

Savile Row [16] is the second tool in the pipeline. It takes as input the model and problem class parameters in Essence Prime, and produces output for a number of different solvers. Savile Row instantiates the model and performs optimisations before translating the instance into the input language of a solver. Currently Savile Row translates to CP solvers Minion [51] and Gecode [52], the learning CP solver Chuffed [53], SAT solvers such as Glucose [54], MaxSAT solvers such as Open-WBO [55], and SMT solvers such as Yices [56], Z3 [57], and Boolector [58].

Once a solution has been found Savile Row translates the solution back into Essence Prime. Conjure then translates the Essence Prime solution back into Essence. Thus the user of Conjure can specify a problem in terms of abstract types such as partition, and receive solutions in terms of the same types.

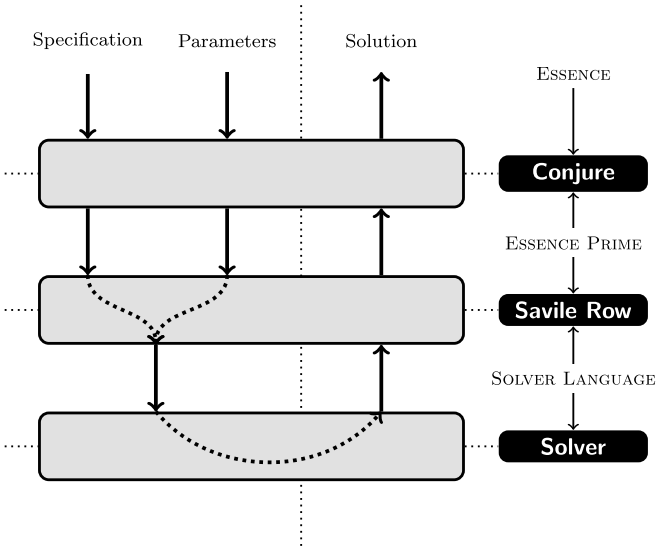


Fig. 8. Automated Constraint Modelling Pipeline.

3.4. *How* Essence *is represented in* Conjure

Problem specifications are represented internally using an abstract syntax tree (AST). A complete specification contains a language declaration line and a list of statements. Each statement is either a declaration (of parameters, decision variables, or aliases), a constraint, an objective (for optimisation problems) or a where statement. Decision variables (find), parameters (given) and aliases (letting) have names as part of their declaration statement and they can be referred to by their name in the subsequent statements. Constraints (such that) and where statements contain a list of Boolean expressions. The objective statement contains a single expression of type `int` or an enumerated type. A problem specification can have at most one objective statement. There is no restriction on the order of statements of different kinds, the only restriction is that declarations cannot be referred to before they are declared, thus circular definitions are disallowed.

Expressions in the Conjure AST are composed of references to existing declarations, operator applications, literal values for the various types in Essence, quantified expressions, and comprehensions. Conjure implements 76 operators in its latest version. We do not give a list of all operators here, these are available in the Conjure documentation [49]. Quantified expressions and comprehensions are commonly found in many modelling languages. Internally, only comprehensions are represented and quantified expressions are converted to comprehensions directly after parsing. Conjure implements a full evaluator for Essence, which can be used to validate solutions. The full evaluator is able to compute a Boolean value for constraint expressions as long as values for the declarations referenced in the expression are fully defined. Typically values for `given`s come from a parameter file and values for `find`s come from solution files during solution validation. In addition to the full evaluator, a partial evaluator is implemented which is used to simplify expressions where possible. The partial evaluator is applied in a very similar way to the refinement rules (discussed in Section 4). The partial evaluator has the highest precedence, so expressions are always evaluated rather than refined if possible.

4. Refinement Rules in Conjure

Conjure translates an abstract problem specification written in Essence into a concrete model in Essence Prime via a series of transformations. These transformations are written as rules in Conjure. There are two main kinds of rules: *representation selection* and *expression refinement*. Applying representation selection rules to each abstract variable in a specification corresponds to choosing a *viewpoint* for the problem. A viewpoint is a selection of variables with associated domains sufficient to characterise the solutions to the problem. Different viewpoints give rise to fundamentally different models of a problem [59,60]. Multiple representation selection rules may be applied to the same abstract variable to create a channelled model [32], in which a single abstract decision variable is refined in multiple ways. Expression refinement rules rewrite expressions to use one of the selected representations of an abstract variable. Thus the two types of rules correspond to modelling steps taken by human modellers: selection of a viewpoint or viewpoints, and formulating the constraints.

Refinement rules in Conjure encode known modelling transformations that are well established in the literature and are known to be correct. We do not formally prove the correctness of the refinement rules; a full and formal exposition of the rules together with proofs of correctness is out of the scope of this paper.

4.1. Representation Selection Rules

Representation selection rules operate on decision variables or parameters with abstract domains. When a representation selection rule is applied to a domain, it removes the outermost abstract type and replaces it with a concrete type such as a matrix. The output domain is not necessarily concrete, however a concrete domain can always be reached by repeated application of representation selection rules.

In some cases the output domain of a representation selection rule may have values in its domain that do not correspond to values of the input domain. In this case, *structural constraints* are needed to rule out these values.

As an example, consider the *Occurrence* representation of a set. The original domain is `set (size n) of T`, where `T` represents an Essence domain. The new domain has one Boolean variable for each value that may be in the set, where the Boolean is assigned true if the value is in the set. The rule is represented below.

```
input-declaration:  find x : set (size n) of T
output-declaration: find x_Occurrence : matrix indexed by [T] of bool
structural-constraint: (sum i : T . toInt(x_Occurrence[i])) = n
```

The `input-declaration` part of the rule is pattern-matched against the abstract domains. The `output-declaration` gives the resulting domain, where the value of `T` is given from the `input-declaration`. Finally the `structural-constraint` requires that `n` of the Booleans are true, as the set is required to be size `n`.

Whenever multiple representation selection rules match one abstract domain, one or more representations must be selected in some way. In Section 6 below we present a simple heuristic that is often able to select a good model.

Each representation selection rule has associated mapping functions that translate between values in the input domain and those in the output domain. The mapping functions are used to translate parameter values from Essence to Essence Prime, and to translate solutions expressed in Essence Prime to Essence (Fig. 8). Each representation only encodes one step of this translation and Conjure applies them successively to convert between Essence and Essence Prime.

4.1.1. Conditional Structural Constraints

Structural constraints are essential for the correctness of representation selection rules. However, in some cases we need to condition the application of these structural constraints on other parts of the model. For example, if the *Occurrence* representation of a set (shown above) were contained in another set of cardinality 0 or 1, then the structural constraint would be required when the outer set has cardinality 1, otherwise the *Occurrence* representation is unused and its structural constraint is not required. For a further example, see Section 4.1.3.

We introduce an operator `structuralCons(X)` representing the structural constraints (if any) of the chosen representation of `X`. Concrete types have no structural constraints and by default these are treated as the `true` constraint. Structural constraints are always applied for the outermost type of the abstract domain of a declaration. Representation selection rules are responsible for applying structural constraints to any abstract decision variables that they declare in their output declaration section. Many representation selection rules simply apply `structuralCons(X)` for every `X` they declare, but some do not. Section 4.1.2 and Section 4.1.3 have examples of representation selection rules that use the `structuralCons` function.

4.1.2. Modelling Symmetry

Symmetry enters constraint models in two ways. Some problems have inherent symmetries, for example the rotations of a chessboard, which if not broken are reflected in the model. Many symmetries however are introduced by the modelling process; in this case a single solution to the problem corresponds to multiple assignments to the variables of the model. For example, in the *Explicit* representation a set is represented as a list — reordering the members of this list does not change the set represented. Frisch et al. [33] show how each representation selection rule of Conjure can be extended to generate a description of the symmetries it introduces and how the generated descriptions can be composed to form a description of the symmetries introduced into the model. However, they do not show how to convert model symmetry descriptions into symmetry breaking constraints.

Conjure takes a different approach to generate symmetry breaking constraints: rules that introduce symmetries also generate a constraint to break those symmetries (excepting unnamed types, discussed below). A modelling symmetry is introduced whenever the application of a representation selection rule increases the number of solutions. This occurs when the output domain, constrained by the structural constraints, has more values than the input domain. Suppose we define the *Explicit* representation of a set as follows.

```
input-declaration:  find x : set (size n) of T
output-declaration: find x_Explicit :
                        matrix indexed by [int(1..n)] of T
structural-constraint: allDifferent(x_Explicit),
    forAll i : int(1..n) . structuralCons(x_Explicit[i])
```

In this rule the `allDifferent` structural constraint prevents repeated values in `x_Explicit`, however it does not constrain the order of the values in the matrix. The structural constraint suffices for correctness, however the rule would

introduce modelling symmetry, which in turn may degrade the performance of a solver. The second line of the structural constraint section applies the structural constraints of the inner type to all elements of this set. Each representation is responsible for applying the structural constraints to the nested objects, since these are not always applied unconditionally. In Section 4.1.3 we see an example of the conditional application of the structural constraints for the elements.

To avoid modelling symmetry, in addition to ensuring the elements are all different we also impose an order on the matrix. As the elements of the matrix can be any type `T` we introduce two new operators, $\dot{\leq}$ and $\dot{<}$ (also written as `.<=` and `.<`). These operators provide a total ordering (and a strict version of the same total ordering) for all types in Essence. These orderings are not intended to be "natural" and are not available in the Essence language. As these orderings are only used to break symmetries, the specific ordering used will never change the solutions of any specification. The two arguments of $\dot{\leq}$ and $\dot{<}$ must have the same representation. They are used only in refinement rules to generate effective symmetry-breaking constraints. Using these orderings, the *Explicit* rule for sets is modified to break all the symmetries it introduces, as follows.

```
input-declaration:  find x : set (size n) of T
output-declaration: find x_Explicit :
                         matrix indexed by [int(1..n)] of T
structural-constraint:
  forAll i : int(1..n-1) . x_Explicit[i] .< x_Explicit[i+1],
  forAll i : int(1..n) . structuralCons(x_Explicit[i])
```

Rather than introducing a chain of $\dot{\leq}$ constraints, this rule exploits the fact that the elements of the set are required to be different and strengthens the ordering to a $\dot{<}$ constraint.

As well as providing efficient and composable symmetry breaking, breaking symmetry immediately in this way has other advantages. Expression refinement rules (described below) can exploit the fact that symmetry breaking is performed immediately to produce more efficient refinements. Consider refining the constraint $S = T$ by representing the sets S and T of the same fixed size as matrices S' and T' with the `allDifferent` structural constraint. To check if S' and T' represent the same set we need to check if each element of S' is equal to *any* element of T', since the order of elements in the matrices can be different. However, with the $\dot{<}$ ordering we can refine $S = T$ to $S' = T'$, because each assignment of S corresponds to exactly one assignment to S'. This gives a much smaller and simpler refined expression and both provides a simpler constraint and smaller search trees.

Both $\dot{\leq}$ and $\dot{<}$ are entirely removed within Conjure by translating them into lexicographic (lex) ordering constraints [61, 62]. The ordering imposed by $\dot{\leq}$ and $\dot{<}$ is allowed to differ depending on the representation chosen for each variable, to allow Conjure to use the most simple and efficient lex ordering constraints. Removing $\dot{\leq}$ and $\dot{<}$ operators is achieved with a small set of rewriting rules. First, references to abstract decision variables are replaced with their representation. If the representation has multiple output declarations (e.g. a matrix and a size variable) then they are contained in a tuple. Once the arguments of the $\dot{\leq}$ or $\dot{<}$ contain no abstract types, each matrix is flattened into a one-dimensional matrix using `flatten`, and each tuple is concatenated into a single one-dimensional matrix using `concatenate`. Finally $A \dot{\leq} B$ is replaced by $A \leq_{lex} B$ and similarly for $\dot{<}$ (or $\leq$ and $<$ for a single integer or Boolean). The `flatten` and `concatenate` functions exist in Essence Prime so there is no need to further translate them.

The representation selection rules in Conjure are designed to avoid introducing modelling symmetry. Many representation selection rules have additional structural constraints to prevent modelling symmetry arising. In this way we maintain a model that is free of modelling symmetry throughout the refinement process with one exception: unnamed types.

Unnamed type symmetry cannot be handled in the same way as the other modelling symmetries introduced by Conjure. Unnamed type symmetries must be removed first, because we cannot put a complete ordering on an unnamed type, or any type which contains an unnamed type, as by definition unnamed types are not ordered. However, breaking general unnamed type symmetry is extremely difficult. Consider the type `set of set (size 2) of U` for an unnamed type `U` – this type represents an undirected graph on a set of vertices `U`, and checking if two graphs are the same (allowing reordering of the vertices) is the famous "Graph Isomorphism" problem, whose complexity is unknown. Extending to two unnamed types with `matrix indexed by [U1,U2] of bool` produces a matrix where the rows and columns can be permuted, which is known to be NP-complete and there have been several papers investigating the best way to partially deal with this symmetry group [63,64]. As a final example, the type `matrix indexed by [int] of U` has value symmetry which can be broken in polynomial time for some problem classes [65]. In future work we will look at general methods of dealing with unnamed type symmetry, which will cover all the different symmetries which can arise from the use of unnamed types.

4.1.3. Types with Variable Size

Many domains in Essence have values of different sizes. A simple example would be a `set` domain with no attributes restricting the size of the set. If the set is a decision variable then deciding the size of the set becomes part of the decision problem. The *Explicit* representation selection rule only works for fixed cardinality sets, whereas variable cardinality sets are also commonly found in combinatorial optimisation problems. We define a representation (called *Explicit-VariableSize*) which uses a single integer decision variable to track the cardinality of the set, and creates a matrix that has sufficient entries of type `T` to represent the largest possible set.

```
input-declaration:  find x : set of T
output-declaration: find x_ExpVarSize : matrix indexed by
                                        [int(1..Tsize)] of T
output-declaration: find x_Card       : int(0..Tsize)
structural-constraints:
  forAll i : int(2..Tsize) . i <= x_Card ->
          x_ExpVarSize[i-1] .< x_ExpVarSize[i],
  forAll i : int(1..Tsize) . i <= x_Card ->
          structuralCons(x_ExpVarSize[i])
```

In this rule, `Tsize` is the smallest of the size of the domain `T` (which is calculated automatically) or the `maxsize` annotation for `x`, if one is given. The structural constraint for Explicit-VariableSize orders the elements for the first `x_Card` indices of the matrix, breaking the modelling symmetry on those elements. However, the remaining elements of the matrix are now free to take any value in `T`, therefore the representation has *conditional symmetry* [66]. Solvers may search over all possible assignments, both increasing the size of the search and producing many solutions which represent the same solution of the specification. Other abstract types that have a non-trivial refinement (multiset, function, relation, and partition) may also be of variable size, so this issue of unconstrained variables occurs in many representations. In general, `dontCare` constraints are used to fix the values of any free variables introduced by the representations. Another example of free variables occur in the representation of partial functions. In cases where a value is not defined in the function, the corresponding image variables are free.

We introduce a new operator named `dontCare` to break conditional symmetry caused by free variables. For any Essence decision variable `x`, `dontCare(x)` assigns all decision variables in the concrete representation of `x` to their smallest value. This prevents the target solver from searching on any of the decision variables in the concrete representation of `x`.

All `dontCare` operators are removed before the Essence Prime model is produced, so there is no need to extend other tools to support it. Removing `dontCare` operators is achieved with a small set of rewriting rules. A `dontCare` operator on a decision variable with an abstract domain is rewritten as `dontCare` on the representation of the decision variable. When `dontCare` is applied to a tuple or a matrix, it is rewritten to apply to each element of the tuple or matrix separately. When `dontCare` is applied to a Boolean or integer variable, it is rewritten to an equality constraint fixing the variable to its smallest value. These rules suffice to remove `dontCare` completely before an Essence Prime model is produced.

The assignment made by `dontCare(x)` may not correspond to a value in the abstract domain of `x`. For example, if the abstract domain is `set (minSize 2) of int(1..3)` and the *Occurrence* representation is used (as in Section 2.4), the current implementation of `dontCare(x)` assigns all variables to `false`, and therefore produces an empty set. This will conflict with the annotation `minSize 2`. Therefore the structural constraints of `x` will conflict with `dontCare(x)`, and we ensure that Conjure avoids asserting both together.

Representation rules are required to ensure each abstract variable they introduce will have exactly one of `dontCare` or `structuralCons` placed on them in any assignment, to ensure both the removal of symmetries and correct answers. The Explicit-VariableSize rule is therefore written as follows:

```
forAll i : int(1..Tsize) . i > x_Card ->
                           dontCare(x_Explicit[i]),
forAll i : int(1..Tsize) . i <= x_Card ->
                           structuralCons(x_Explicit[i])
```

The `dontCare` constraint is refined using the standard expression refinement processes within Conjure, and it is used in some refinements of several other abstract types. In Section 5 we evaluate the impact of breaking conditional symmetry using `dontCare`.

4.1.4. 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 (e.g. [63]). 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 specification 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 symmetry or 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 in the original specification either. Conjure only breaks the symmetry introduced by a representation selection rule. For this purpose, it posts symmetry breaking constraints on the concrete decision variables it generates. The concrete variables are not present in the original specification so it is impossible to write conflicting constraints in terms of them.

Refining any Essence specification using Conjure produces a model that has an identical number of solutions to the specification. Therefore we have broken all symmetries which would lead to one Essence solution mapping to multiple

Essence Prime solutions. We only need to ensure each representation selection rule in isolation preserves exactly one assignment for each solution, and the application of any set of representation selection rules will also preserve the number of solutions.

We have focused in this paper on breaking modelling 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, for example representing the reflections and rotations of a chessboard. Assuming the symmetry in the specification has been detected (a topic not addressed in this paper) and broken consistently by adding constraints to the specification prior to refinement (for example via the lex leader method [61]) there will be no consistency issue with the way in which Conjure breaks modelling symmetry.

4.1.5. Viewpoint Selection

Choosing a representation selection rule to apply to a decision variable corresponds to a human modeller selecting a viewpoint. It is therefore crucially important to the efficiency of the model, affecting the ease of stating constraints, their propagation and ultimately the efficiency of the search for a solution.

Conjure makes all representation choices in one pass, separating the choice of representations from the actual application of the representation selection rules. It chooses a representation for each decision variable in the specification. Every reference to a decision variable is tagged with the name of its representation (which guides the application of expression refinement rules, as described in Section 4.2 below). For simplicity we assume here that each decision variable has one representation. However, in a channelled model a decision variable may have multiple representations. Section 4.3 describes how Conjure generates channelled models.

4.1.6. Representation Selection Rules in Conjure

Table 5 gives a brief description of each of Conjure's representation selection rules. There are 17 representations in total, spread across the 6 abstract domain constructors. Three representations (*FunctionAsRelation*, *RelationAsSet*, *PartitionAsSet*) work by converting an abstract domain to another abstract domain, which is then converted to a concrete domain by subsequent representation rule applications. We briefly explain the remaining representations in this section.

A common method shared by several representations is to use marker or flag variables to indicate the relevant members of a matrix. For example, in a variable size set representation (with a marker variable), Conjure creates a matrix with sufficient entries to represent the maximum number of elements of the set. In addition a marker variable is used to indicate the size of the set. Decision variables in the matrix that are not used are irrelevant to the final value of the abstract variable. These are fixed to break symmetry using `dontCare` constraints, as described in Section 4.1.3.

There are two main kinds of representations for sets: the *Occurrence* representation and four flavours of explicit representations. The *Occurrence* representation creates a Boolean variable for every potential member of the set. This representation does not introduce modelling symmetry, but it can create a prohibitively large number of variables when given a large set domain. The basic *Explicit* representation works for fixed cardinality set variables. For a set with cardinality n, it creates a matrix indexed by $\{1..n\}$ where each element of the matrix represents one member of the set. Symmetry breaking constraints ensure the matrix is in increasing order. The *ExplicitVariableSizeMarker* and *ExplicitVariableSizeFlags* representations work for variable cardinality sets. They both have a matrix similar to *Explicit* but with one matrix element per potential element of the set, using the maximum cardinality of the set as the limit. The former then uses a single integer variable to denote the cardinality of the set and the latter uses a Boolean variable per element of the matrix to indicate membership. Appropriate symmetry breaking constraints are added to enforce an increasing order among the elements of the set and to fix the irrelevant variables using `dontCare` constraints. *ExplicitVariableSizeDummy* is similar to *Explicit* but adds a dummy value to the domain of the elements of the matrix.[3]

Representations of multisets are similar to those of sets. In contrast to sets, multisets allow repeated values. In order to accommodate this, the multiset *Occurrence* representation introduces an integer decision variable (instead of a Boolean) for each value of the set. The domain of this variable ranges from zero to the maximum number of occurrences allowed per value. The *ExplicitFlags* representation uses a decision variable per distinct value and a corresponding decision variable for the number of repetitions of that value. The *ExplicitRepetition* representation uses a matrix of decision variables bounded by the maximum cardinality of the multiset. Repeated values are allowed in this matrix and the resulting symmetry is broken by placing them in non-decreasing order.

Sequences in Essence are ordered collections of values of variable length (with an upper bound). Sequences are represented with a matrix and a length variable. Elements of the matrix which have an index greater than the length of the sequence are fixed using `dontCare` constraints.

There are four representations of functions. *FunctionAsMatrix* represents a total function $\tau_1 \to \tau_2$ using a matrix indexed by τ_1, containing τ_2. The remaining three representations are used for partial functions. The *FunctionAsMatrixPartial* is the *FunctionAsMatrix* representation plus a Boolean variable corresponding to every value in τ_1 to indicate whether this value is defined in the function. Undefined values are fixed using `dontCare` constraints. *FunctionAsMatrixDummy* extends *FunctionAsMatrix* with a dummy value to indicate undefined values.

[3] Some solvers support decision variables with 'set of int' domains. In addition to the representation options presented here, Conjure could trivially be made to output these variables without converting them to matrices.

Table 5
Representation selection rules in Conjure.

Representation	Description
`set of` τ	
Occurrence	Matrix of Boolean flags indicating presence of each value.
Explicit	Fixed size matrix of distinct values.
ExplicitVariableSizeMarker	Explicit with size variable.
ExplicitVariableSizeFlags	Explicit with Boolean flags to mark unused elements.
ExplicitVariableSizeDummy	Explicit with dummy value for unused elements.
`mset of` τ	
Occurrence	Matrix of integers indicating frequency of each value.
ExplicitFlags	Matrix of distinct values with counter for each.
ExplicitRepetition	Matrix of values, repetition allowed.
`sequence of` τ	
ExplicitBounded	Matrix of τ and length variable.
`function` τ_1 `->` τ_2	
FunctionAsMatrix	Total functions only. Matrix indexed by τ_1 of τ_2.
FunctionAsMatrixPartial	Matrix indexed by τ_1 of τ_2 and matrix of Boolean flags to mark unused elements.
FunctionAsMatrixDummy	Matrix indexed by τ_1 of τ_2 extended with a dummy value to indicate missing values.
FunctionAsRelation	Relation where each element of τ_1 is related to at most one element of τ_2.
`relation of (` $\tau_1, \ldots, \tau_i, \ldots$ `)`	
RelationAsMatrix	For relations of integers or Booleans only. Matrix of Boolean flags indicating presence of each tuple in the relation.
RelationAsSet	Set of tuples.
`partition from` τ	
Occurrence	For partitions of integers only. Cells are numbered. A matrix indexed by τ contains the cell number of each element. The size and first element of each cell are also represented.
PartitionAsSet	Set of sets where inner sets represent cells of the partition.

The *RelationAsMatrix* representation has a Boolean matrix indexed by the components of the relation, where a true value indicates relation membership. The partition (from τ) *Occurrence* representation has a matrix indexed by τ to represent the cell of the partition that each value belongs to. Cells are identified using integers. The cardinality and the first element of each cell are also represented for efficiency reasons. The modelling symmetry arising from this representation is broken by its structural constraints.

4.2. Expression Refinement Rules

Expression refinement rules are the second kind of rules in Conjure. They are used to translate Essence expressions to equivalent Essence Prime expressions. They may or may not depend on the representations of decision variables and parameters. Rules that do not depend on representations are called horizontal rules, and those that do are called vertical rules. Horizontal rules do not change the representation of decision variables, they merely translate Essence expressions to other Essence expressions. Horizontal rules are representation independent, and they reduce the need for a very large number of representation-dependent vertical rules.

4.2.1. Vertical Rules

Vertical rules replace references to abstract decision variables with their representations. There must exist vertical rules for the most basic operations on the abstract types. One of the most important classes of vertical rules are the comprehension generator rules that allow comprehensions to iterate over elements contained in an abstract decision variable. Suppose we have the following comprehension containing the abstract variable `S`, of type `set of int`. All items in `S` must be odd. In addition we have an `in` operator, one of the simplest binary operators on sets.

```
find S : set (size 3) of int(1..10)
such that and([ i%2 = 1 | i <- S ]), 5 in S
```

If the *Occurrence* representation is chosen for `S`, one vertical rule replaces the generator `i <- S` with `i: int(1..10), i in S`. Another vertical rule replaces both `i in S` and `5 in S` as follows.

```
find SOccurrence : matrix indexed by [int(1..10)] of bool
such that
  sum([ SOccurrence[i] | i : int(1..10) ]) = 3,
  and([ i%2 = 1 | i : int(1..10), SOccurrence[i] ]),
  SOccurrence[5]
```

If the *Explicit* representation is chosen, the resulting model is quite different. A vertical rule replaces the generator `i <- S` with `q: int(1..3)`, and references to `i` with `SExplicit[q]`, producing a straightforward model of the first

constraint. For the second constraint, there is no vertical rule so a horizontal rule is applied first, producing `or([q = 5 | q <- S])`. From there, the same vertical rule is applied to the generator `q <- S`, producing the model below. To complete the refinement of this model fragment the `.<` would be refined as described in Section 4.1.2.

```
find SExplicit : matrix indexed by [int(1..3)] of int(1..10)
such that
  and([ S_Explicit[q] .< S_Explicit[q + 1] | q : int(1..2)]),
  and([ SExplicit[q]%2 = 1 | q : int(1..3) ]),
  or( [ SExplicit[q] = 5 | q : int(1..3) ])
```

All comprehension generators `i <- T` have a vertical rule for every possible representation of `T`, and all abstract types are allowed in a generator (see Section 3.1). Other operators may have vertical rules for some representations and not others. In the example above, `i in S` was refined with a vertical rule when `S` took the *Occurrence* representation. Vertical rules take priority over horizontal rules.

4.2.2. Horizontal Rules

Horizontal rules are entirely independent of the chosen representation of the abstract decision variables. They allow Conjure to reformulate expressions, adding to the diversity of models that Conjure can produce and also avoiding the need for a huge number of vertical rules. When there is no vertical rule available for an expression, Conjure applies a horizontal rule to replace the expression with a simpler expression, often by decomposing an operator. Repeated application of horizontal rules always allows Conjure to reach a vertical rule.

For example, suppose we have two decision variables `A` and `B` of type `set of int`, and one constraint `A = B`. Refinement of equality is important for channelling constraints (as described in Section 4.3 below) and for cases where equality is part of a larger expression.

```
find A, B : set (size 3) of int(1..10)
such that A = B
```

Suppose the *Occurrence* representation is chosen for `A` and *Explicit* is chosen for `B` in the constraint `A = B`. There is no vertical rule for equality between these two distinct representations of sets. A horizontal rule is applied to decompose `A = B` into `A subsetEq B /\ B subsetEq A`. However `subsetEq` also has no vertical rule. Another horizontal rule is applied to each of the `subsetEq` operators, resulting in the following specification.

```
find A, B : set (size 3) of int(1..10)
such that (and([ i in A -> i in B | i : int(1..10) ]) /\
           and([ i in B -> i in A | i : int(1..10) ]))
```

To complete the refinement, Conjure applies the relevant vertical rules for `in`, replacing `i in A` with `AOccurrence[i]`, and `i in B` with the following:

```
exists q : int(1..3) . BExplicit[q]=i
```

Thanks to being representation-oblivious, horizontal rules allow Conjure to achieve full coverage of the Essence language using a manageable number of rules and without having to repeat similar rules for each new representation.

4.3. Channelling Multiple Representations

Combining multiple representations of one abstract decision variable in a channelled model can be remarkably powerful [67]. Constraints may be stated on the most appropriate of the chosen representations, allowing for more concise expression of constraints and in some cases improved propagation, both of which can improve efficiency of the search for a solution. However channelling also introduces overheads in the form of additional decision variables and constraints, which may outweigh their potential benefits.

Conjure chooses a representation for each *reference to* a decision variable in the specification, therefore it may choose multiple representations for one decision variable. All representation choices are made in one pass where every reference to a decision variable is tagged with the name of a suitable representation. In this way the choice of representations is separated from the actual application of the representation selection rules.

When a decision variable is represented in more than one way, channelling constraints are added to ensure consistency between the representations. A channelling constraint is simply an equality between two references to the same decision variable, where the two references are tagged with different representations. The equality is then refined using the standard refinement processes for expressions, described in Section 4.2. A channelling constraint is created for every pair of distinct representations. An example of refining a channelling constraint for the knapsack problem is given in Section 2.5.

By default Conjure produces multiple models by enumerating all possible ways of selecting representations (i.e. all ways of tagging every reference to a decision variable in the AST) and all possible ways of generating constraint expressions once a representation is selected. Depending on the specification, large numbers of models may be produced from one specification (as shown in Table 7). We discuss the issue of selecting an effective model in Section 6, and in Section 7 we evaluate Conjure by examining whether it can generate known good models from the literature for a wide range of specifications.

Table 6
Number of solutions with and without `dontCare` constraints. A $\geq$ indicates number of solutions found within 1 hour CPU timeout.

Outer \ Inner	`set`		`multiset`		`function`		`relation`		`partition`	
	With	Without	With	Without	With	Without	With	Without	With	Without
`set`	11	38	22	87	46	632	67	297	15	845
`multiset`	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

5. The Impact of Breaking Modelling Symmetries

In this section we evaluate the impact of breaking model symmetries automatically. Throughout, it is important to bear in mind that these symmetries are broken by Conjure at the problem class level, hence the benefit of symmetry breaking is automatically obtained for every instance of the problem class being refined. This approach is substantially more efficient than analysing an individual instance to identify symmetries [68]. At present there are two mechanisms for breaking modelling symmetries. The first breaks unconditional variable symmetries using ordering constraints (introduced in Section 4.1.2). In this case the number of symmetries can be represented with closed-form expressions and we give a detailed example in Section 5.1. The second mechanism breaks conditional symmetries that arise when parts of representations are unused in a solution (introduced in Section 4.1.3). Here the number of symmetries, and thus the impact of symmetry-breaking constraints, is not straightforward to represent mathematically so we have an experiment in Section 5.2 below.

5.1. Breaking Unconditional Variable Symmetries

In order to illustrate both the importance of symmetry breaking, and the way in which the high level of abstraction of Essence allows us to avoid the expensive step of detecting modelling symmetries, we will consider the refinement of the Social Golfers Problem specification presented in Fig. 1. The single abstract decision variable in the specification is a set (representing the weeks) of partitions (representing the groups of golfers). A standard refinement of a fixed-cardinality set, particularly when its elements are themselves complex objects, is into a matrix with the same number of elements as the cardinality of the set (the *Explicit* representation). Of course, since matrices have indices whereas sets do not, this immediately introduces a symmetry whereby any permutation of the matrix represents the same set. In this case, there will be $w!$ such symmetries. However, the refinement rule employed by Conjure recognises this modelling symmetry and breaks it as it enters the model, by ordering the elements of the matrix, without the need for a costly symmetry-identification process in the final model.

The partition of the golfers can be thought of as a set of sets of golfers subject to the additional constraints that the outer set contains exactly g sets, each of size s, and the intersection of any pair of inner sets is empty. A natural refinement of this nested object is into a $g \times s$ matrix, introducing a symmetry on the $g!$ possible arrangements of the groups and the $s!$ arrangements of golfers within those groups. Since each group can be arranged independently, this results in $g!(s!)^g$ symmetries for each partition, which again Conjure identifies and breaks as they enter the model.

Since each partition forms one of the weeks, the final model derived as above has $w!g!(s!)^g$ symmetries in total. This is a vast number for even relatively small instances of the social golfers problem, which, if left in the model, could have a significant adverse influence on the performance of search. Conjure's ability to deal with these symmetries automatically at the class level (as described in Section 4.1.2) is therefore very valuable.

5.2. Breaking Conditional Symmetries

Conditional symmetries arise when refining abstract domains where the values have distinct sizes, and so parts of the representation may be redundant in a solution (depending on the chosen representation). As described in Section 4.1.3, `dontCare` constraints are used to assign redundant variables in order to break the conditional symmetry. We ran an experiment 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 experiment also demonstrates that arbitrary combinations of nested types can be handled, even with conditional symmetries in each. In these experiments Savile Row and Minion were run with their default settings 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 Essence domain constructors for the other layers. The outer two layers have a bounded size of 2, so can also be empty or size 1, meaning that both layers will require conditional symmetry breaking using `dontCare` constraints. Moreover, the structural constraints of the inner layer will need to be posted conditionally. 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 chose one model per problem using the CompactEP heuristic (see Section 6).

Table 6 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. The most extreme cases involve partitions, and can produce hundreds of millions of solutions when there are only ten symmetrically distinct ones. When using `dontCare` constraints, these symmetric solutions are avoided and the solver need not waste effort searching through them.

6. Model Selection with the CompactEP Heuristic

Conjure is able to produce multiple models by enumerating all possible ways of selecting representations. If time is limited it is sensible to provide a rapid model selection method, avoiding both generating all models and training using instance data. In earlier work we proposed a method based on racing [34] to select a subset of the models that perform well on a given set of training instances. Racing methods allow comparing alternative algorithms without necessarily having to run all algorithms on all instances. Racing for model selection can be very computationally expensive. The focus of this paper is on refinement within Conjure so we omit model selection methods that are essentially external to Conjure such as racing.

Conjure contains greedy model selection heuristics that are used for making local decisions during model generation. These can be employed during both representation selection and expression refinement. The default heuristic is called CompactEP, which stands for "compact except parameters", and it is a combination of the Compact heuristic and the Sparse heuristic. We define these heuristics in the following.

The Compact heuristic favours transformations that produce simpler types of variables and smaller expressions at each point during refinement where multiple rules are applicable. We define the *compact* ordering on abstract types as follows: concrete domains (such as `bool`, `matrix`) are smaller than abstract domains; within concrete domains, `bool` is smaller than `int` and `int` is smaller than `matrix`. These rules are applied recursively, so that a one-dimensional matrix of `int` is smaller than any two-dimensional matrix. Abstract type constructors have the ordering `set < mset < sequence < function < relation < partition`, which is also applied recursively. At each stage of representation selection, the CompactEP heuristic will select the smallest domain according to this order. As an example, a `set(size n) of int` is represented as a `matrix indexed by [int] of bool` with the *Occurrence* representation, and as a `matrix indexed by [int] of int` with the *Explicit* representation. As `bool` is smaller than `int` under our ordering, Compact will always pick the *Occurrence* representation in this example.

During expression refinement Compact chooses the rule that produces the most shallow abstract syntax tree (AST) directly following its application. For example an expression like `a subsetEq` has a shallower AST (depth 1) than `forAll i in a. exists j in b. i = j` (depth 3). To break ties, an arbitrary total ordering is defined over all abstract syntax trees.

The Sparse heuristic is intended to enable small representations of parameter values. It employs a built-in ordering of representations that gives priority to those that take advantage of sparsity. For example, the *Explicit* representation would take priority over the *Occurrence* representation for a fixed-cardinality set because *Explicit* scales with the cardinality whereas *Occurrence* scales with the number of values potentially in the set. Consider a parameter with the domain `relation of (int(1..100) * int(1..100))`. A sparse member of this domain like `relation((1,2), (3,4))` would require 10,000 Booleans with the *RelationAsMatrix* representation and only 4 integers with the *RelationAsSet* representation.

The default CompactEP heuristic is a combination of these two heuristics: during representation selection, Conjure uses the Sparse heuristic when representing problem class parameters and the Compact heuristic for everything else.

7. Evaluation: Conjure Produces Kernels of Good Models

Conjure provides full coverage of the Essence language. It has at least one variable representation rule (typically several, see Table 5) for every abstract variable type, and horizontal and vertical expression refinement rules for all the operators defined on them. In this section we test the hypothesis that the *kernels* of constraint models written by experts can be automatically generated by refining a problem's abstract specification. For two CP models to have the same model kernel, they need to share the same viewpoint, the same representation of decision variables and the same formulation of the problem constraints, together with symmetry breaking. Expert models can have additional features such as implied constraints or dominance breaking [69] constraints but these are not considered to be in the kernel of the CP model for this evaluation. Some expert models contain global constraints that are not present in Essence Prime. In these cases, if Conjure generates an equivalent decomposition then we consider the two models to have the same kernel.

In order to test this hypothesis, we took a diverse set of 42 benchmark problems drawn from the literature and refined them with Conjure. Our main source for these problems is CSPLib [48]. We cover the entire CSPLib problem class collection (at the time of writing), except those problems that are naturally represented using only matrices of Booleans or integers, i.e. without the facilities that Essence provides in addition to those of lower level constraint modelling languages.

In Table 7 we present the set of problem classes and the abstract types of their decision variables in Essence. We also cite the papers that contain a kernel that Conjure is able to generate. We begin by noting the variety of decision variable types involved in the benchmark problems, representing further evidence that the current collection of rules, the rewrite

Table 7
Running Conjure on 42 benchmark problems from CSPLib. We highlight the features of Essence used in the problem specification for each problem class and include a reference to at least one published model for each problem that is comparable to one of the models automatically generated by Conjure. In addition, we present the estimated number of models Conjure can produce using 6 configurations of the model selection heuristics.

#	Problem	Essence features	Refs	Pruning			No Pruning		
				NoCh	VarsCh	FullCh	NoCh	VarsCh	FullCh
1	Car Sequencing	function	[71]	1	1	1	4	64	10,368
2	Template Design	1 1D function, 1 2D function	[72]	1	1	1	12	576	5184
3	Quasigroup Existence	2D function	[73]	1	1	1	3	729	729
5	Low Autocorrelation Binary Sequences	function variable	[74]	1	1	1	4	64	64
6	Golomb Ruler	set variable	[75]	2	32	32	2	32	32
7	All-Interval Series	2 bijective functions	[76]	1	1	1	16	1024	1024
8	Vessel Loading	4 functions	[77]	1	1	4	256	$1.8 \cdot 10^{19}$	$4.8 \cdot 10^{23}$
9	Perfect Square Placement	2 functions	[78]	1	1	1	16	$1.7 \cdot 10^{7}$	$1.7 \cdot 10^{7}$
10	Social Golfers Problem	set of partition	[31]	3	9	9	3	9	9
13	Progressive Party Problem	set of partition	[79]	4	256	256	16	$2.6 \cdot 10^{5}$	$9.4 \cdot 10^{6}$
15	Schur's Lemma	partition	[80]	4	16	32	5	25	50
16	Traffic Lights	function	[81]	1	1	1	3	27	27
17	Ramsey Numbers	function	[82]	2	128	512	3	2187	8748
18	Water Bucket Problem	2 sequences	[35,83]	1	1	32	4	$1.1 \cdot 10^{9}$	$1.1 \cdot 10^{12}$
21	Crossfigures	sequence, variant	[84]	1	1	1	1	1	1
22	Bus Driver Scheduling	partition of int	[85]	4	64	256	9	729	2916
24	Langford's number problem	sequence and injective function	[86,87]	1	1	1	3	243	243
26	Sports Tournament Scheduling	relation between enums a set of enums	[88]	2	16	16	2	16	16
28	Balanced Incomplete Block Designs	relation of unnamed types	[89]	1	1	1	3	243	243
30	Balanced Academic Curriculum Problem	binary relation and a function variable	[90]	1	1	1	4	16,384	$2.9 \cdot 10^{5}$
31	Rack Configuration Problem	a partial nested function	[91]	1	1	1	12	$2.5 \cdot 10^{5}$	$5.4 \cdot 10^{8}$
32	Maximum Density Still Life	set of tuples	[92–94]	2	256	256	2	256	256
33	Word Design for DNA Computing	set of function	[95]	2	32	32	8	32768	32768
34	Warehouse Location Problem	function	[27]	4	4	4	16	1024	18432
36	Fixed Length Error Correcting Codes	set of functions	[64]	1	1	2	4	64	192
38	Steel Mill Slab Design	partition of orders.	[96,97]	4	$2.6 \cdot 10^{5}$	$2.6 \cdot 10^{5}$	8	$1.3 \cdot 10^{8}$	$3.6 \cdot 10^{9}$
39	The Rehearsal Problem	3 functions, 1 bijection	[98]	1	1	1	64	$1.7 \cdot 10^{7}$	$1.2 \cdot 10^{9}$
40	Wagner-Whitin Distribution Problem	partial 2D function	[99]	1	1	4	1	1	4
44	Steiner triple systems	matrix of set	[100]	2	8	8	2	8	8
45	The Covering Array Problem	mset of function	[101]	2	4	8	8	64	320
49	Number Partitioning	2 set variables	[102]	4	256	256	4	256	256
51	Tank Allocation	set parameters	[103]	1	1	32	1	1	32
53	Graceful Double Wheel Graphs	injective functions	[104]	1	1	1	16	$2.7 \cdot 10^{8}$	$2.7 \cdot 10^{8}$
53	Graceful Gears	injective functions	[104]	1	1	1	16	65536	65536
53	Graceful Helms	injective functions	[104]	1	1	1	16	$4.2 \cdot 10^{6}$	$4.2 \cdot 10^{6}$
53	Graceful Wheel Graphs	injective functions	[104]	1	1	1	16	65536	65536
55	Equidistant Freq. Permutation Arrays	set	[105]	1	1	1	4	256	256
56	Synchronous Optical Networking	mset of set	[106]	8	512	1024	8	512	1024
65	Optimal Financial Portfolio Design	set of set	[107]	2	4	4	2	4	4
83	Transshipment Problem	2 partial functions	[108]	4	256	256	16	65,536	$2.1 \cdot 10^{7}$
85	Van der Waerden Numbers	partition of numbers	[109,110]	4	16	16	5	25	25
86	Capacitated Vehicle Routing Problem	set of sequences	[111]	4	256	256	4	256	256
110	Peaceably Co-existing Armies of Queens	2 sets of tuples	[112]	8	2048	4096	60	$1.5 \cdot 10^{11}$	$7.4 \cdot 10^{12}$
115	Tail Assignment	relations, sets of nested functions	[113]	6	$3 \cdot 10^{14}$	$3 \cdot 10^{14}$	45	$5.2 \cdot 10^{30}$	$4.2 \cdot 10^{32}$
116	Vellino's Problem	partial function of msets	[114]	1	1	1	4	64	10368

rule mechanism, and the Conjure system as a whole is capable of refining a wide variety of abstract problem specifications into concrete models. The number of models generated for a problem specification depends on the number of representation options for its decision variables.

7.1. Configurations of Conjure

In Table 7 we report a lower bound on the number of models that can be generated by Conjure with six configurations. One source of variation in models is the selection of different representations for decision variables, parameters, and quantified variables. We calculate the exact number of representations available for each by examining the domain. A second source of variation arises from expression refinement. We calculate a lower bound on the number of attainable models by taking the product of the number of representation options for each reference to a declaration in the model (where channelling is enabled for the declaration). For example, if a decision variable may be channelled, has two representations, and is referred to three times in the model, there are $2^3 = 8$ ways of tagging the references with a representation. The result of this calculation is a lower bound because it ignores the potential for multiple expression refinement pathways after the selection of representations.

The six configurations represent different trade-offs between time taken and the ability to generate diverse models. One option is to prune the set of representations:

- *Pruning*: Use a built-in heuristic to filter the list of representations. This heuristic only allows the use of one variable cardinality representation (*ExplicitVariableSizeMarker*) for sets created by the *RelationAsSet* and *PartitionAsSet* representations.
- *No Pruning*: Explore all applicable representations for every decision variable and parameter.

Pruning and *No Pruning* are combined with each of the following options for channelling:

- *NoCh*: Decision variables and parameters each have only one representation (no channelling).
- *VarsCh*: Channelling is allowed for decision variables but not for parameters.
- *FullCh*: Channelling is allowed for both decision variables and parameters.

The number of models Conjure can generate for a problem class depends heavily on the abstract types used in the problem specification. In particular, decision variables and parameters that have abstract domains present an opportunity for using different representations. When channelling is enabled, the number of models also depends on the number of times each decision variable (or parameter) is mentioned in the constraints: each use of a decision variable presents an opportunity for a new representation. In addition to choosing one representation for each use of decision variables, we allow the addition of one extra representation, used mainly for providing a search order [70]. In Table 7 we present the numbers of models for the six configurations. In terms of the numbers of models, it is always the case that $\mathit{NoCh} \leq \mathit{VarsCh} \leq \mathit{FullCh}$, and $\mathit{Pruning} \leq \mathit{No\ Pruning}$. In some cases channelling dramatically increases the number of models (Steel Mill Slab Design for example), and similarly turning off pruning can have a dramatic impact (for example, the Water Bucket Problem).

7.2. Comparing Generated Models to Published Models

In this section we briefly compare the models generated by Conjure to published models written by expert modellers for each of the 42 problem classes.

CSPLib 1 For the car sequencing problem using the default heuristic Conjure generates a model that uses an integer matrix to represent the function variable. This is the same viewpoint as the model published in [71]. In addition, Conjure generates a 2-dimensional Boolean representation of the same function variable which is commonly used when developing MIP or SAT models.

CSPLib 2 The template design problem has two function variables in its Essence specification. These variables are represented using integer matrices via the default heuristic since they are total functions. This model has the same viewpoint as [72].

CSPLib 3 For several variations of the quasigroup existence problem, Zhang [73] used a 2-dimensional matrix to represent an integer square grid. This model is produced as the default model by Conjure.

CSPLib 5 The low autocorrelation binary sequences problem contains a function variable which is modelled in a similar way to the template design problem (CSPLib 2), see [74].

CSPLib 6 The golomb ruler problem is naturally modelled as a set. From this set model Conjure generates an explicit representation with symmetry breaking and a Boolean occurrence representation. In the explicit model, the distinctness of the inter-tick distances are modelled using an all-different constraint (thanks to Savile Row). The explicit model is given in Section 2 of [75] and the occurrence model in Section 7 of the same paper.

CSPLib 7 The all-interval series problem is modelled as 2 bijective functions in Essence. Previous work on this model looks into breaking symmetry [76] and they use a 1-dimensional array-based representation for each function variable with appropriate constraints to enforce the bijection property. The default model generated by Conjure is the same as this model.

CSPLib 8 The vessel loading problem is described by Brown [77]. He used an array based model to represent the function variables that are found in the Essence problem specification. Among other representations, Conjure generates the same viewpoint as this published model.

CSPLib 9 The perfect square placement problem is specified using function variables in Essence. The default model generated by Conjure uses the same viewpoint as the model given in [78]. This published model uses the cumulative global constraint which is not generated by Conjure currently, instead Conjure generates an equivalent decomposition.

CSPLib 10 The social golfers problem is specified using a set of partitions in Essence. The set represents the weeks and each partition is the schedule for a week. This abstract domain gives rise to a 3-dimensional matrix model, with appropriate constraints posted to enforce the set and the partition structure. This is the viewpoint used by the default model generated by Conjure and it corresponds to the model presented in [31].

CSPLib 13 Progressive party problem includes a set of partitions in its problem specification. This domain can be refined into representations that have the same viewpoint as both of the models presented in [79].

CSPLib 15 Schur's lemma [80] is specified using a single partition variable. Using quantification over all sets of triples of potential members of this partition and an *apart* operator, the problem is stated using a single top level constraint. The default model Conjure generates uses a set of sets and includes automated symmetry breaking constraints.

CSPLib 16 The traffic lights problem is used as an example in [81] as a demonstration of higher-arity constraints. In Essence this problem can be specified using functions and set membership.

CSPLib 17 The Ramsey numbers problem is modelled using a function variable and universal quantification over fixed size subgraphs. The default model generated by Conjure uses a variable to represent each edge in the graph and its colour, which corresponds to the model presented in [82].

CSPLib 18 The water bucket problem is a planning problem. It is modelled using a sequence of states and actions in Essence. The sequence type allows modelling a list with a bounded (but not fixed) length. The default model generated for this problem represents the states and the actions explicitly and breaks the conditional symmetry [35] arising from the variable length of the data structures. A comparable model is given in [83].

CSPLib 21 The crossfigures problem benefits from bounded length sequences in a similar way to the water bucket problem. In addition, it uses a variant type to represent the 8 different kinds of clues succinctly. The generated Essence Prime model is comparable to a MiniZinc model published on the CSPLib page [84].

CSPLib 22 The natural language specification of the bus driver scheduling problem starts with the following sentence: "Bus driver scheduling can be formulated as a set partitioning problem". The problem specification in Essence is very succinct and has a single decision variable that represents this partition. Starting from this problem specification, one of the models generated by Conjure uses the same core viewpoint as the model published in [85].

CSPLib 24 A channelled Essence model for the Langford's number problem is presented in [86]. The output of Conjure for this problem corresponds very closely to the models presented by Smith [87].

CSPLib 26 The sports tournament scheduling problem is specified using an arity 3 relation of week, period and a set of two teams. Each entry in the relation indicates that the two teams are timetabled to play a game on the selected week and the period. This abstract type (and using the relation projection operator) allows a succinct specification of the problem in Essence. One of the generated CP models uses a viewpoint similar to the one published in [88].

CSPLib 28 The balanced incomplete design problem (BIBD) is typically modelled in CP using a 2-dimensional array with symmetry-breaking constraints [89]. In Essence, the problem is specified using a relation with arity 2 between two unnamed types. This domain allows us to break all of the symmetry in this problem (which is not the most computationally efficient approach), or produce the more commonly used double-lex constraints.

CSPLib 30 The balanced academic curriculum problem (BACP) is modelled using a relation to represent the prerequisites between courses and a function variable to represent the assignment of periods to courses. Using this problem specification, Conjure generates a model very similar to the one published in [90].

CSPLib 31 The rack configuration problem is modelled using a partial function that represents whether each rack is used in a solution or not, and if it is, the model and quantity of each card type in this rack. This representation captures the decision abstractly and allows the generation of the kernels of both models presented in [91].

CSPLib 32 For the maximum density still life problem, we obtain a comparable model to that of Bosch & Trick [92] with the difference that Conjure chooses an explicit instead of an occurrence representation. However, we obtain nothing like the dual model obtained by Smith [93] in which supercells in the original grid are used as variables: this is an example of a reformulation that could be implemented generally but remains outside the scope of this paper. Using even more advanced techniques, a complete solution to the problem has been found for all n [94].

CSPLib 33 The DNA word design problem is very succinctly specified in Essence using a set of functions. It is an optimisation problem where the objective is to minimise the cardinality of the abstract set. For this problem, Conjure produces a model similar to the one published in [95], together with symmetry breaking constraints between the (function) members of the set.

CSPLib 34 The warehouse location problem is a typical network flow problem. Function variables in Essence can be used to specify this problem at a high level of abstraction, and Conjure is able to generate the two main alternative viewpoints described in [27] as well as channelled versions of these two viewpoints.

CSPLib 36 The fixed length error correcting codes problem uses a partial function in its Essence problem specification and an abstract decision variable whose domain is a set of functions. Thanks to this abstract type, the models produced by Conjure include symmetry breaking constraints similar to those presented in [64].

CSPLib 38 For the steel mill slab design problem, Conjure obtains models where the set of orders assigned to each slab, and the set of colours on each slab, are refined using occurrence or explicit representations. If occurrence is used for both, then the model is similar to the model in [96]. The later model of [97] uses a different viewpoint (as well as exploiting dominance) and is not generated by Conjure.

CSPLib 39 The rehearsal problem uses 3 function variables, one of which is a bijection. When defining its objective function, this problem specification uses the list comprehension feature of Essence. Smith [98] presents a model which is very similar to one of the models generated by Conjure.

CSPLib 40 The distribution problem with Wagner-Whitin costs is a warehouse stock distribution problem where each warehouse may order stock from one other warehouse at each time step. The specification uses a partial function variable to represent the orders. Conjure generates the *conventional* viewpoint of Tarim and Miguel [99] where value 0 is used to represent no order, and a variation of it where additional Boolean variables indicate whether orders are made, however Conjure does not generate the *echelon* model [99].

CSPLib 41 The n-fractions puzzle is not a very complex problem to specify, but it still benefits from a surjective attribute in Essence. Frisch et al. [115] use this problem to explore implied constraints and one of the models generated by Conjure is the same as the one they present.

CSPLib 44 The Steiner triple systems problem is a special case of the balanced incomplete block design problem. Its problem specification uses set variables and the set intersection operator and Conjure is able to generate an explicit representation of these set variables, similar to a viewpoint used in [100].

CSPLib 45 The covering array problem is modelled using a 2-dimensional matrix variable. The statement of the constraints uses a quantified expression over all values of a fixed length sequence. The output models are similar to those published in [101].

CSPLib 49 The set partitioning problem is very directly specified in Essence using 2 set variables and a constraint to enforce the main sum constraint in the problem. Alternatively a partition with 2 parts can also be used. In either case, the output models will use an *Explicit* set representation based viewpoint or an *Occurrence* based set representation. Depending on instance size one or the other model is likely to be a better choice. Both of these viewpoints are explored in previous work in the context of mathematical programming [102].

CSPLib 51 Schaus et al. [103] present a viewpoint for the tank allocation problem that uses a single integer variable representing the product type per tank. The Essence problem specification follows this viewpoint closely, benefiting from Essence features to represent the parameters (a set of sets for representing *incompatibilities*) and when stating the constraints.

There are 4 variants of the graceful graphs problem on **CSPLib 53**: Wheel Graphs, Double Wheel Graphs, Gears, Helms. Smith and Puget [104] present two viewpoints for this family of problems: one primarily based on the nodes, and another primarily based on the edges. The Essence problem specification gives rise to viewpoints based on the nodes.

CSPLib 55 The equidistant frequency permutation arrays (EFPA) problem is specified using a single set variable. This allows Conjure to generate several alternative models, with symmetry breaking and channelling between representations. The generated models include the Boolean, Non-Boolean, and Channelled models that were presented in [105].

CSPLib 56 The synchronous optical networking (SONET) problem uses a single variable with a nested domain: multiset of set of nodes. In addition to the declaration of this variable, the problem specification has a single statement for the objective and a single statement for the problem constraint (to enforce that the demand is met). Starting from this high level problem specification, Conjure not only produces a model comparable to the one published in [106], but it also creates the same symmetry breaking constraints automatically.

CSPLib 65 The optimal financial portfolio design problem is specified in Essence using a set of set of integers as a single top level decision variable. The main constraint is written very succinctly using the universal subset quantification feature of Essence. Starting from this high-level specification, Conjure generates a model that uses the same matrix-based viewpoint that is published in [107].

CSPLib 83 The transshipment problem [108] is a network flow problem where the nodes are warehouses, transshipment points, or customers. The model in Essence uses a partial function from pairs of nodes to the amount of flow on the corresponding edge to model this structure. The default model generated by Conjure uses a 2-dimensional Boolean matrix to model edge existence and a 2-dimensional integer matrix to model the amount of flow on an edge. A second model which uses a list of triples (2 nodes and the flow amount) is also generated. The latter model is likely to be a good choice for sparse networks.

CSPLib 85 There is one common model for Van der Waerden numbers (using a 2-dimensional Boolean matrix viewpoint) used in both CP [109] and SAT [110]. This model is among the models generated by Conjure and it is chosen by the default heuristic.

CSPLib 86 There are two main approaches to modelling the capacitated vehicle routing problem: vehicle flow formulations and set partitioning formulations [111]. The first has an integer variable per edge representing the flow on that edge, and this is the default model produced by Conjure. The second uses Boolean variables to represent set partitioning, and Conjure also generates this model.

CSPLib 110 The Essence problem specification for the 'peaceable armies of queens' problem uses two set variables to represent the location of white and black queens. These sets are represented by Conjure in several ways, including the viewpoint given in [112], together with the symmetry breaking constraints presented there.

CSPLib 115 The tail assignment problem is defined using a single function variable. This function variable finds a partial mapping from flights to flights, representing a route, for every plane. The basic model in [113] uses 3 sets of decision variables to represent the same information. Conjure generates a comparable viewpoint automatically.

CSPLib 116 The Essence problem specification for Vellino's problem uses a partial function of multisets to represent the contents of each active bin. Bins that are not used are undefined in this function. The problem is stated using function operators (defined, range, quantification etc). A similar viewpoint to the default model generated by Conjure is published in [114].

In this section we have demonstrated that Conjure is able to generate models that are similar to published models produced by experts, for a wide range of problem classes drawn from a public repository. Also, all six of the abstract types in Essence (set, multiset, sequence, function, relation, and partition) are used in the specifications of the 42 problem classes, showing that each of these types is a necessary part of Essence. Moreover, in 30 out of these 42 problem classes Conjure's default heuristic is able to choose a model that is equivalent to a published model for the same problem class.

8. Related Work in Automated Constraint Modelling by Refinement

This section primarily surveys other languages and systems that have been employed in refinement-based approaches to automated constraint modelling, and compares them with our own work on Essence and Conjure. Beyond this body of work, there exists a variety of other approaches to automated modelling, which we discuss briefly before proceeding.

One such line of work is example driven. O'Casey [23] is a case based reasoning tool, which uses recordings of previous problem solving episodes. Problems are paired with problem instances to form a *case*. The experience obtained from cases are mainly the selection of propagators and search heuristics. Conacq [116] is a SAT-based version space algorithm to acquire constraint networks. Its inputs are the set of decision variables, and a collection of positive and negative examples. Positive examples are valid solutions to the problem and negative examples are non-solutions. It automatically generates constraints by applying machine learning techniques. The Constraint Seeker [117] focuses on the automated acquisition of individual global constraints from a large collection of positive and negative examples. The Model Seeker [7] uses only positive examples to learn complete models.

Another approach is to transform an existing constraint model to improve solver performance. The CGrass [118] system explores the idea of reformulating CP models using a collection of rules in order to improve them. It is limited to integer variables, and arithmetic and logical operators on integer expressions, and does not change representations of decision variables, but it can rearrange constraint expressions and reduce domains of decision variables. Tailor [11] performs common-subexpression elimination (CSE) and its successor, Savile Row [16], extends CSE and adds other powerful transformations.

MiniZinc [12] is a medium-level constraint modelling language. It contains features common to many CP modelling languages such as Boolean and integer domains, and arrays for collections of these variables. MiniZinc can be used to describe problem class models, however it does not perform any reformulations at the class level. When presented with problem instance data, the class model is instantiated into an instance model which can be targeted to one of several solver backends. MiniZinc uses a solver-dependent instance level language called FlatZinc to interact with solvers.

8.1. Refining Abstract Constraint Problems

The NP-Spec language [119] allows the specification of NP-complete problems in a subset of existential second order logic. It provides a small number of high level domains, sets and partitions of integers, which are automatically refined into decision variables with simpler domains. NP-Spec provides only one way to refine each high-level domain and operator. Hence, it does not allow for the generation of alternative models.

The ESRA language [25] has a particular focus on decision variables with relation domains. It is translated to the language OPL [114,13], a constraint modelling language with similar facilities to Essence Prime, by refining relation domains and operators. Like NP-Spec, however, it does not consider multiple alternative refinement pathways. Moreover, the abstract domains offered by ESRA cannot be nested arbitrarily.

The F language [27] supports function variables. Problems modelled in F are refined into OPL using a system called Fiona. F supports function attributes such as *total* and *bijective*. Function domains in F cannot be nested arbitrarily, a function variable is simply a mapping between non-nested domains like integers or enumerations. Fiona does, however, support multiple alternative refinements for function domains, among which it selects using a number of heuristics. Fiona always generates a single output model using these heuristics. If the same function variable is refined in multiple ways within a single model, Fiona is able to generate channelling constraints automatically.

The constraint language most closely resembling Essence is Zinc [28]. Both languages support type constructors that can be nested to arbitrary depth, and they have a number of type constructors in common, such as sets, arrays and tuples. Essence supports more abstract decision variables than Zinc, for example via multiset, partition, function, and relation type constructors, which in Zinc must be modelled using a constrained collection of variables of a more primitive type.

Quantification over decision variables, which Essence supports, is vital for concision when dealing with variables that have nested domains. Essence and Zinc provide a similar selection of atomic types, although Zinc supports floats, which Essence does not, and unnamed types [120] are unique to Essence. Zinc is extensible via user-defined functions and predicates, a feature which Essence lacks.

Work on refining Zinc has focused on the production of models for different solving paradigms, such as mixed integer programming, constraint programming, and local search [121,29,30], rather than alternative refinement pathways for a particular type of solver. De Koninck et al. describe plans to use annotations to guide the use of alternative refinements manually [29].

Hernández [122] considers the problem of channelling different representations of high-level variables. She produces similar results to the implementation used in Conjure, which produces channelling constraints by refining $X = X$ where the left and right X have different representations.

8.2. Encoding Constraint Models to Other Formalisms

A related body of work seeks to encode a given constraint model into another formalism, such as mixed integer programming (MIP), propositional satisfiability (SAT), or SAT modulo theories (SMT). In selecting how the variables and constraints of the constraint model are to be encoded, this approach shares many of the concerns of the refinement of abstract specifications described above. The substantial difference is in the lower level of abstraction of the input.

One popular method of solving constraint problems is to encode them to SAT and employ a SAT solver. The two key considerations are: first, the way in which the CP variables (i.e. variables of the constraint model) are encoded by a set of SAT variables and associated clauses; and second, how the constraints are encoded into SAT clauses (and additional SAT variables if necessary). The simplest such scheme introduces one SAT variable per domain value of each CP variable [123], adding clauses to ensure that every CP variable takes exactly one value. An important alternative is the *log* encoding [124], in which we represent a variable of domain size n by $\lceil \log_2 n \rceil$ SAT variables. Another alternative is the *order* encoding: each SAT variable indicates whether the CP variable is greater than a constant [125–127]. The order encoding can be useful when inequality reasoning is important. There is an extensive literature on encoding constraints into SAT, including generic encodings of arbitrary constraints (e.g. the *direct* or *support* encodings among others [123,124,128]). Special-purpose encodings for particular constraint types can vastly outperform generic encodings, for example *cardinality networks* for counting constraints [129], and the *compact order encoding* for sums [130] among many others. Picat is notable for using a log encoding throughout [131]. The availability of many encodings suggests that automatic selection of encodings is important. It is complementary to automatic generation and selection of models. Proteus [132], meSAT [133] and Satune [134] are examples of systems that automatically select SAT encodings.

SMT solvers have made remarkable progress in recent years, making SMT an attractive target for encoding constraint models. FZN2OMT [135] translates the FlatZinc language to SMT, while SR-SMT [136] is a component of Savile Row [16] that outputs SMT. Both systems are able to target multiple theories and multiple SMT solvers. Selection of encodings is similarly important when encoding to SMT as it is with SAT.

Encoding of constraint modelling languages to MIP (*linearization*) has a long history. OPL is an early example [137], however the OPL system was not able to linearize the entire OPL language. More recently, Rafeh and Jaberi [30] presented LinZinc, a library to linearize the Zinc language in its entirety, and Belov et al. [138] presented a linearization of MiniZinc. For several constraint types (such as allDifferent), linearization of the constraint requires a 0/1 variable for each domain value of each CP variable in scope, similar to the *direct* SAT encoding. In contrast to SAT and SMT, the issue of selection among multiple encodings is not discussed in any of these works [137,30,138].

A related field in which specification languages are extensively used is that of formalising computing systems. Specification languages such as Z [139] and VDM-SL [140] are used for describing general computing systems. These languages typically allow lambda expressions, set theoretic operators, and first-order logic. In comparison, Essence is a problem specification language in the domain of combinatorial problem solving and it offers specific features such as decision variables, a rich selection of finite domains and operators for posting a variety of constraints on these decision variables. Generic formal specification languages do not typically encode decision problems, instead they encode properties of a system that are required to be true, and enable formal proofs of those properties.

9. Conclusions and Future Work

In this paper we have presented the automated constraint modelling system Conjure. It employs a set of refinement rules to transform the specification of a parameterised problem class in the abstract constraint specification language Essence into a concrete constraint model. By varying the selection and application of these rules Conjure can produce a set of alternative models. We have demonstrated on a large set of problem classes that, in the vast majority of cases, the set produced includes those formulated by human experts in the literature. Furthermore, we have presented a heuristic by which an effective model can be selected.

A particular advantage of this approach is in the treatment of symmetry. Much of the symmetry typically present in a constraint model arrives through the process of modelling [33]. Conjure recognises and removes this symmetry as it enters a model, removing the need for an expensive symmetry detection step following model formulation, as used by

other approaches [141,37]. Furthermore, the symmetry breaking constraints added to the model are valid for the entire problem class, rather than just a single instance. An important item of future work is the treatment of symmetry arising from unnamed types.

Another important item of future work is a more informed method of model selection to complement the data-free heuristic presented herein. Following the practice of algorithm selection [142], a set of training instances for a problem class could be used to learn how to select an effective model for an unseen instance from the same problem class.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] F. Rossi, P. Van Beek, T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.
[2] E.C. Freuder, Progress towards the holy grail, Constraints 23 (2) (2018) 158–171.
[3] Z. Kiziltan, M. Lippi, P. Torroni, Constraint detection in natural language problem descriptions, in: IJCAI, 2016, pp. 744–750.
[4] L. De Raedt, A. Passerini, S. Teso, Learning constraints from examples, in: Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence, AAAI, New Orleans, USA, 2018, pp. 02–07.
[5] C. Bessiere, F. Koriche, N. Lazaar, B. O'Sullivan, Constraint acquisition, Artif. Intell. 244 (2017) 315–342.
[6] R. Arcangioli, C. Bessiere, N. Lazaar, Multiple constraint aquisition, in: IJCAI: International Joint Conference on Artificial Intelligence, 2016, pp. 698–704.
[7] N. Beldiceanu, H. Simonis, A model seeker: extracting global constraint models from positive examples, in: 18th International Conference on Principles and Practice of Constraint Programming, 2012, pp. 141–157.
[8] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, T. Walsh, Constraint acquisition via partial queries, in: IJCAI'2013: 23rd International Joint Conference on Artificial Intelligence, 2013, p. 7.
[9] C. Bessiere, R. Coletta, A. Daoudi, N. Lazaar, Boosting constraint acquisition via generalization queries, in: Proc. of the 21st European Conference on Artificial Intelligence, 2014, pp. 99–104.
[10] K. Shchekotykhin, G. Friedrich, Argumentation based constraint acquisition, in: 2009 Ninth IEEE International Conference on Data Mining, 2009, pp. 476–482.
[11] A. Rendl, Thesis: Effective Compilation of Constraint Models, Ph.D. thesis, University of St, Andrews, 2010.
[12] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, G. Tack, MiniZinc: towards a standard {CP} modelling language, in: Proceedings of 13th International Conference on Principles and Practice of Constraint Programming {CP-2007}, 2007, pp. 529–543.
[13] P. Van Hentenryck, The OPL Optimization Programming Language, MIT Press, Cambridge, MA, USA, 1999.
[14] P. Mills, E.P.K. Tsang, R. Williams, J. Ford, J. Borrett, {EaCL} 1.5: an Easy Abstract Constraint Optimisation Programming Language, Tech. Rep., University of Essex, Colchester, UK, dec 1999.
[15] P. Nightingale, Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination, in: 20th International Conference on Principles and Practice of Constraint Programming (CP 2014), Springer, 2014, pp. 590–605.
[16] P. Nightingale, Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, P. Spracklen, Automatically improving constraint models in Savile Row, Artif. Intell. 251 (2017) 35–61.
[17] P. Nightingale, P. Spracklen, I. Miguel, Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row, in: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015), Springer, 2015, pp. 330–340.
[18] A.M. Frisch, I. Miguel, T. Walsh, Cgrass: a system for transforming constraint satisfaction problems, in: Recent Advances in Constraints, Springer, 2003, pp. 15–30.
[19] S. Colton, I. Miguel, Constraint generation via automated theory formation, in: International Conference on Principles and Practice of Constraint Programming, Springer, Berlin Heidelberg, 2001, pp. 575–579.
[20] J. Charnley, S. Colton, I. Miguel, Automatic generation of implied constraints, in: ECAI, Vol. 141, 2006, pp. 73–77.
[21] C. Bessiere, R. Coletta, T. Petit, et al., Learning implied global constraints, in: 20th International Joint Conference on Artificial Intelligence, 2007, pp. 44–49.
[22] K. Leo, C. Mears, G. Tack, M.G. De La Banda, Globalizing constraint models, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2013, pp. 432–447.
[23] J. Little, C. Gebruers, D.G. Bridge, E.C. Freuder, Using case-based reasoning to write constraint programs, in: Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}, 2003, p. 983.
[24] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, I. Miguel, The rules of constraint modelling, in: L.P. Kaelbling, A. Saffiotti (Eds.), Proc. of the IJCAI 2005, Professional Book Center, 2005, pp. 109–116, http://www.ijcai.org/papers/1667.pdf.
[25] P. Flener, J. Pearson, M. Ågren, Introducing esra, a relational language for modelling combinatorial problems, in: M. Bruynooghe (Ed.), LOPSTR 2003, in: Lecture Notes in Computer Science, vol. 3018, Springer, 2003, pp. 214–232.
[26] A.M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel, Essence: a constraint language for specifying combinatorial problems, Constraints 13 (3) (2008) 268–306, http://link.springer.com/article/10.1007/s10601-008-9047-y.
[27] B. Hnich, Thesis: function variables for constraint programming, AI Commun. 16 (2) (2003) 131–132.
[28] K. Marriott, N. Nethercote, R. Rafeh, P.J. Stuckey, M.G. de la Banda, M. Wallace, The design of the zinc modelling language, Constraints 13 (3) (2008) 229–267, https://doi.org/10.1007/s10601-008-9041-4.
[29] L.D. Koninck, S. Brand, P.J. Stuckey, Data independent type reduction for zinc, in: Proceedings of the 9th International Workshop on Reformulating Constraint Satisfaction Problems, 2010.
[30] R. Rafeh, N. Jaberi, LinZinc: a library for linearizing zinc models, Iran. J. Sci. Technol. Trans. Electr. Eng. 40 (1) (2016) 63–73.
[31] M. Sellmann, W. Harvey, Heuristic constraint propagation–using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem, in: CPAIOR'02, Citeseer, 2002.
[32] B. Cheng, J.H.-M. Lee, J. Wu, Speeding up constraint propagation by redundant modeling, in: International Conference on Principles and Practice of Constraint Programming, Springer, 1996, pp. 91–103.
[33] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, I. Miguel, Symmetry in the generation of constraint models, in: Proceedings of the International Symmetry Conference, 2007.

[34] O. Akgun, A.M. Frisch, I.P. Gent, B.S. Hussain, C. Jefferson, L. Kotthoff, I. Miguel, P. Nightingale, Automated symmetry breaking and model selection in Conjure, in: Proceedings of 19th International Conference on Principles and Practice of Constraint Programming (CP 2013), 2013, pp. 107–116.
[35] O. Akgun, I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Breaking conditional symmetry in automated constraint modelling with Conjure, in: Proceedings of the 21st European Conference on Artificial Intelligence (ECAI), 2014, pp. 3–8.
[36] T. Mancini, M. Cadoli, Detecting and breaking symmetries by reasoning on problem specifications, in: International Symposium on Abstraction, Reformulation, and Approximation, Springer, 2005, pp. 165–181.
[37] C. Mears, T. Niven, M. Jackson, M. Wallace, Proving symmetries by model transformation, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2011, pp. 591–605.
[38] C. Mears, M.G. de la Banda, M. Wallace, B. Demoen, A method for detecting symmetries in constraint models and its generalisation, Constraints 20 (2) (2015) 235–273.
[39] A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, T. Walsh, Towards csp model reformulation at multiple levels of abstraction, in: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems, 2002, pp. 42–56.
[40] A.M. Frisch, I. Miguel, T. Walsh, Refining abstract specifications of constraint satisfaction problems, in: Proceedings of the Tenth Workshop on Automated Reasoning, 2003, pp. 29–31.
[41] A. Bakewell, A.M. Frisch, I. Miguel, Towards automatic modelling of constraint satisfaction problems: a system based on compositional refinement, in: Notes of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, CP-03 Post-Conference Workshop, 2003.
[42] A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, T. Walsh, Transforming and refining abstract constraint specifications, in: 6th Symposium on Abstraction, Reformulation and Approximation, Springer, 2005, pp. 76–91.
[43] A.M. Frisch, M. Grum, C. Jefferson, B.M. Hernández, I. Miguel, The sssence of Essence, in: Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, 2005, pp. 73–88.
[44] A.M. Frisch, M. Grum, C. Jefferson, B.M. Hernández, I. Miguel, The design of essence: a constraint language for specifying combinatorial problems, in: IJCAI, Vol. 7, 2007, pp. 80–87.
[45] O. Akgun, A.M. Frisch, B. Hnich, C. Jefferson, I. Miguel, Conjure revisited: towards automated constraint modelling, in: Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation, 2010.
[46] O. Akgun, I. Miguel, C. Jefferson, A.M. Frisch, B. Hnich, Extensible automated constraint modelling, in: W. Burgard, D. Roth (Eds.), AAAI 2011 - Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011, AAAI Press, 2011.
[47] O. Akgun, Extensible automated constraint modelling via refinement of abstract problem specifications, Ph.D. thesis, University of St Andrews, 2014.
[48] I.P. Gent, T. Walsh, {CSPLib}: A Problem Library for Constraints, 2005.
[49] Özgür Akgün, A. Salamon, Conjure documentation, release 2.3.0, arXiv:1910.00475, 2019.
[50] P. Nightingale, A. Rendl, Essence' description, arXiv:1601.02865 [cs.AI].
[51] I.P. Gent, C. Jefferson, I. Miguel Minion, A fast scalable constraint solver, in: Proceedings ECAI 2006, 2006, pp. 98–102.
[52] Gecode Team, Gecode: generic constraint development environment, available from http://www.gecode.org, 2006.
[53] G. Chu, P.J. Stuckey, A. Schutt, T. Ehlers, G. Gange, K. Francis, Chuffed, available from, https://github.com/chuffed/chuffed/, 2018.
[54] G. Audemard, L. Simon, Predicting learnt clauses quality in modern {SAT} solvers, in: IJCAI, 2009, pp. 399–404.
[55] R. Martins, V. Manquinho, I. Lynce Open-wbo, A modular maxsat solver, in: C. Sinz, U. Egly (Eds.), Theory and Applications of Satisfiability Testing – SAT 2014, Springer International Publishing, Cham, 2014, pp. 438–445.
[56] B. Dutertre, Yices 2.2, in: A. Biere, R. Bloem (Eds.), Computer-Aided Verification (CAV'2014), in: Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 737–744.
[57] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[58] A. Niemetz, M. Preiner, A. Biere, Boolector 2.0 system description, J. Satisf. Boolean Model. Comput. 9 (2014) 53–58 (published 2015).
[59] Y.C. Law, J.H.M. Lee, Model induction: a new source of CSP model redundancy, in: Proceedings of the National Conference on Artificial Intelligence, AAAI Press, MIT Press, Menlo Park, CA, Cambridge, MA, London, 1999, 2002, pp. 54–61.
[60] B.M. Smith, Handbook of Constraint Programming, Chapter Modelling, 2006.
[61] J. Crawford, M. Ginsberg, E. Luks, A. Roy, Symmetry-breaking predicates for search problems, KR 96 (1996) (1996) 148–159.
[62] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Global constraints for lexicographic orderings, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2002, pp. 93–108.
[63] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetries in matrix models, in: Proceedings of 8th International Conference on Principles and Practice of Constraint Programming {CP-2002}, 2002, pp. 462–476.
[64] A.M. Frisch, C. Jefferson, I. Miguel, Constraints for breaking more row and column symmetries, in: F. Rossi (Ed.), Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}, in: Lecture Notes in Computer Science, vol. 2833, Springer, Berlin Heidelberg, 2003, pp. 318–332.
[65] P. van Hentenryck, P. Flener, J. Pearson, M. Agren, Tractable symmetry breaking for CSPs with interchangeable values, in: G. Gottlob, T. Walsh (Eds.), Proc. IJCAI'03, Morgan Kaufmann, 2003, pp. 277–282.
[66] I.P. Gent, T. Kelsey, S. Linton, I. McDonald, I. Miguel, B.M. Smith, Conditional symmetry breaking, in: P. van Beek (Ed.), Proceedings of 11th International Conference on Principles and Practice of Constraint Programming {CP-2005}, in: Lecture Notes in Computer Science, vol. 3709, Springer, 2005, pp. 256–270.
[67] B. Cheng, K.M.F. Choi, J.H.-M. Lee, J. Wu, Increasing constraint propagation by redundant modeling: an experience report, Constraints 4 (2) (1999) 167–192.
[68] C. Mears, M.G. de la Banda, M. Wallace, On implementing symmetry detection, Constraints 14 (4) (2009) 443–477.
[69] J.C. Beck, S.D. Prestwich, Exploiting dominance in three symmetric problems, in: Fourth International Workshop on Symmetry and Constraint Satisfaction Problems, 2004.
[70] C.W. Choi, J.H.M. Lee, P.J. Stuckey, Propagation redundancy in redundant modelling, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2003, pp. 229–243.
[71] M. Dincbas, H. Simonis, P. Van Hentenryck, Solving the car-sequencing problem in constraint logic programming, in: Proceedings of the 8th European Conference on Artificial Intelligence (ECAI 1988), 1988, pp. 290–295.
[72] L. Proll, B. Smith, Integer linear programming and constraint programming approaches to a template design problem, INFORMS J. Comput. 10 (3) (1998) 265–275.
[73] H. Zhang, Specifying latin square problems in propositional logic, in: Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, 1997, pp. 115–146.
[74] I. Dotú, P. Van Hentenryck, A note on low autocorrelation binary sequences, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2006, pp. 685–689.
[75] B.M. Smith, K. Stergiou, T. Walsh, Modelling the {Golomb Ruler} problem, in: Proc. of Workshop on Non Binary Constraints (at {IJCAI} 99), 1999.

[76] I.P. Gent, I. McDonald, B.M. Smith, Conditional Symmetry in the All-Interval Series Problem, 2003.
[77] K.N. Brown, Loading supply vessels by forward checking and unenforced guillotine cuts, in: 17th Workshop of the UK Planning and Scheduling SIG, 1998.
[78] H. Simonis, B. O'Sullivan, Search strategies for rectangle packing, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2008, pp. 52–66.
[79] B.M. Smith, S.C. Brailsford, P.M. Hubbard, H.P. Williams, The progressive party problem: integer linear programming and constraint programming compared, Constraints 1 (1–2) (1996) 119–138.
[80] R. Guy, Unsolved Problems in Number Theory, Problem Books in Mathematics / Unsolved Problems in Intuitive Mathematics, Springer, 2004, http://books.google.co.uk/books?id=1AP2CEGxTkgC.
[81] W. Hower, Revisiting global constraint satisfaction, Inf. Process. Lett. 66 (1) (1998) 41–48, https://doi.org/10.1016/S0020-0190(98)00023-4.
[82] I.P. Gent, B. Smith, Symmetry Breaking During Search in Constraint Programming, Citeseer, 1999.
[83] F.A. Azevedo, A prototype for general planning, in: 2005 Portuguese Conference on Artificial Intelligence, IEEE, 2005, pp. 24–32.
[84] T. Walsh, CSPLib problem 021: Crossfigures, http://www.csplib.org/Problems/prob021.
[85] S.D. Curtis, B.M. Smith, A. Wren, Constructing driver schedules using iterative repair, in: Proceedings of Practical Application of Constraint Logic Programming (PACLP), Citeseer, 2000, pp. 59–78.
[86] Ö. Akgün, I. Miguel, Modelling langford's problem: a viewpoint for search, in: Proceedings of the 17th International Workshop on Reformulating Constraint Satisfaction Problems, 2018.
[87] B.M. Smith, Dual models of permutation problems, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2001, pp. 615–619.
[88] A. Schaerf, Scheduling sport tournaments using constraint logic programming, Constraints 4 (1) (1999) 43–65.
[89] P. Meseguer, C. Torras, Exploiting symmetries within constraint satisfaction search, Artif. Intell. 129 (1–2) (2001) 133–163.
[90] B. Hnich, Z. Kiziltan, T. Walsh, Modelling a balanced academic curriculum problem, in: CP-AI-OR-2002, 2002, pp. 121–131.
[91] Z. Kızıltan, B. Hnich, Symmetry breaking in a rack configuration problem, in: Proceedings of IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints. International Joint Conference on Artificial Intelligence, 2001.
[92] R. Bosch, M. Trick, Constraint programming and hybrid formulations for three life designs, Ann. Oper. Res. 130 (2004) 41–56.
[93] B.M. Smith, A dual graph translation of a problem in 'Life', in: Proc. 8th International Conference on the Principles and Practice of Constraint Programming (CP 2002), 2002, pp. 402–414.
[94] G. Chu, P.J. Stuckey, A complete solution to the maximum density still life problem, Artif. Intell. 184–185 (2012) 1–16, https://doi.org/10.1016/j.artint.2012.02.001, http://www.sciencedirect.com/science/article/pii/S0004370212000124.
[95] M. Codish, M. Frank, V. Lagoon, The DNA word design problem: a new constraint model and new results, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 585–591, https://doi.org/10.24963/ijcai.2017/82.
[96] A.M. Frisch, I. Miguel, T. Walsh, Symmetry and implied constraints in the steel mill slab design problem, in: Proc. CP'01 Wshop on Modelling and Problem Formulation, 2001.
[97] A. Gargani, P. Refalo, An efficient model and strategy for the steel mill slab design problem, in: International Conference on Principles and Practice of Constraint Programming, 2007, pp. 77–89.
[98] B.M. Smith, Constraint programming in practice: Scheduling a rehearsal, Research Report APES-67-2003, APES group, 2003.
[99] S.A. Tarim, I. Miguel, Echelon stock formulation of arborescent distribution systems: an application to the wagner-whitin problem, in: International Conference on Integration of Artificial Intelligence (ai) and Operations Research (or) Techniques in Constraint Programming, Springer, 2004, pp. 302–318.
[100] C.J. Colbourn, Embedding partial steiner triple systems is np-complete, J. Comb. Theory, Ser. A 35 (1) (1983) 100–105.
[101] B. Hnich, S.D. Prestwich, E. Selensky, B.M. Smith, Constraint models for the covering test problem, Constraints 11 (2–3) (2006) 199–219.
[102] J. Kojić, Integer linear programming model for multidimensional two-way number partitioning problem, Comput. Math. Appl. 60 (8) (2010) 2302–2308.
[103] P. Schaus, J.-C. Régin, R. Van Schaeren, W. Dullaert, B. Raa, Cardinality reasoning for bin-packing constraint: application to a tank allocation problem, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2012, pp. 815–822.
[104] B.M. Smith, J.-F. Puget, Constraint models for graceful graphs, Constraints 15 (1) (2010) 64–92.
[105] S. Huczynska, P. McKay, I. Miguel, P. Nightingale, Modelling equidistant frequency permutation arrays: an application of constraints to mathematics, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2009, pp. 50–64.
[106] B.M. Smith, Symmetry and search in a network design problem, in: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, 2005, pp. 336–350.
[107] P. Flener, J. Pearson, L.G. Reyna, Financial portfolio optimisation, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2004, pp. 227–241.
[108] B. Hoppe, É. Tardos, The quickest transshipment problem, Math. Oper. Res. 25 (1) (2000) 36–62.
[109] M. Heule, T. Walsh, Symmetry within solutions, in: Proceedings of AAAI, Vol. 10, 2010, pp. 77–82.
[110] M.R. Dransfield, V.W. Marek, M. Truszczyński, Satisfiability and computing van der waerden numbers, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 1–13.
[111] P. Toth, D. Vigo, Models, relaxations and exact approaches for the capacitated vehicle routing problem, Discrete Appl. Math. 123 (1–3) (2002) 487–512.
[112] B.M. Smith, K.E. Petrie, I.P. Gent, Models and symmetry breaking for 'peaceable armies of queens', in: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, 2004, pp. 271–286.
[113] M. Grönkvist, A constraint programming model for tail assignment, in: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, 2004, pp. 142–156.
[114] P. Van Hentenryck, L. Michel, L. Perron, J.-C. Régin, Constraint programming in opl, in: Principles and Practice of Declarative Programming, Springer, 1999, pp. 98–116.
[115] A.M. Frisch, C. Jefferson, I. Miguel, Symmetry breaking as a prelude to implied constraints: a constraint modelling pattern, in: Proc. ECAI 2004, 2004, pp. 171–175.
[116] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan, Acquiring Constraint Networks Using a Sat-Based Version Space Algorithm, Proceedings of the National Conference on Artificial Intelligence, vol. 21, AAAI Press, MIT Press, Menlo Park, CA, Cambridge, MA, London, 1999, 2006, p. 1565.
[117] N. Beldiceanu, H. Simonis, A constraint seeker: finding and ranking global constraints from examples, in: Proceedings of 17th International Conference on Principles and Practice of Constraint Programming {CP-2011}, Springer, 2011, pp. 12–26.
[118] A.M. Frisch, I. Miguel, T. Walsh, CGRASS: a system for transforming constraint satisfaction problems, in: B. O'Sullivan (Ed.), International Workshop on Constraint Solving and Constraint Logic Programming, in: Lecture Notes in Computer Science, vol. 2627, Springer, 2002, pp. 15–30.
[119] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, D. Vasile, {NP-SPEC}: an executable specification language for solving all problems in {NP}, Comput. Lang. 26 (2000) 165–195, http://citeseer.ist.psu.edu/71095.html.

[120] A.M. Frisch, I. Miguel, The Concept and Provenance of Unnamed, Indistinguishable Types, sep 2006.
[121] R. Becket, S. Brand, M. Brown, G.J. Duck, T. Feydy, J. Fischer, J. Huang, K. Marriott, N. Nethercote, J. Puchinger, et al., The many roads leading to rome: solving zinc models by various solvers, in: 7th International Workshop on Constraint Modelling and Reformulation, 2008.
[122] B. Martínez-Hernández, B. Martinez-Hernandez, Thesis: the Systematic Generation of Channelled Models in Constraint Satisfaction, Ph.D. thesis, University of York, 2008.
[123] J. De Kleer, A comparison of atms and csp techniques, in: IJCAI, Vol. 89, Citeseer, 1989, pp. 290–296.
[124] T. Walsh, Sat v csp, in: Principles and Practice of Constraint Programming–CP …(1894), 2000, pp. 441–456, http://link.springer.com/chapter/10.1007/3-540-45349-0_32.
[125] I.P. Gent, P. Nightingale, A new encoding of AllDifferent into SAT, in: Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (CP2004), 2004, pp. 95–110.
[126] C. Ansótegui, F. Manyà, Mapping problems with finite-domain variables into problems with boolean variables, in: SAT 2004 - the Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings, 2004, http://www.satisfiability.org/SAT04/programme/53.pdf.
[127] N. Tamura, A. Taga, S. Kitagawa, M. Banbara, Compiling finite linear CSP into SAT, Constraints. An Int. J. 14 (2) (2009) 254–272, https://doi.org/10.1007/s10601-008-9061-0.
[128] I.P. Gent, Arc consistency in SAT, in: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002), 2002, pp. 121–125.
[129] R. Asín, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, Cardinality networks: a theoretical and empirical study, Constraints 16 (2) (2011) 195–221, https://doi.org/10.1007/s10601-010-9105-0.
[130] T. Tanjo, N. Tamura, M. Banbara, Azucar: a sat-based csp solver using compact order encoding, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2012, pp. 456–462.
[131] N.-F. Zhou, H. Kjellerstrand, Optimizing SAT encodings for arithmetic constraints, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2017, pp. 671–686.
[132] B. Hurley, Exploiting machine learning for combinatorial problem solving and optimisation, Ph.D. thesis, University College Cork, 2016.
[133] M. Stojadinovic, F. Maric, meSAT: multiple encodings of CSP to SAT, Constraints 19 (4) (2014) 380–403, https://doi.org/10.1007/s10601-014-9165-7.
[134] H. Gorjiara, G.H. Xu, B. Demsky, Satune: synthesizing efficient sat encoders, in: Proceedings of the ACM on Programming Languages 4 (OOPSLA), 2020, pp. 1–32.
[135] F. Contaldo, P. Trentin, R. Sebastiani, From minizinc to optimization modulo theories and back (extended version), CoRR, arXiv:1912.01476 [abs], http://arxiv.org/abs/1912.01476.
[136] E. Davidson, O. Akgün, J. Espasa, P. Nightingale, Effective encodings of constraint programming models to SMT, in: Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming, 2020, pp. 143–159.
[137] P.V. Hentenryck, Constraint and integer programming in OPL, INFORMS J. Comput. 14 (4) (2002) 345–372.
[138] G. Belov, P.J. Stuckey, G. Tack, M. Wallace, Improved linearization of constraint programming models, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2016, pp. 49–65.
[139] J. Woodcock, J. Davies, Z. Using, Specification, Refinement and Proof, Prentice Hall International, 1996.
[140] N. Plat, P.G. Larsen, An overview of the ISO/VDM-SL standard, ACM SIGPLAN Not. 27 (8) (1992) 76–82.
[141] T. Mancini, M. Cadoli, Detecting and breaking symmetries by reasoning on problem specifications, in: Abstraction, Reformulation and Approximation, in: Lecture Notes in Computer Science, vol. 3607, Springer, Berlin Heidelberg, 2005, pp. 165–181.
[142] J.R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118.