

Towards Reformulating Essence Specifications for Robustness

Özgür Akgün¹, Alan M. Frisch², Ian P. Gent¹, Christopher Jefferson¹,
Ian Miguel¹, Peter Nightingale², and András Z. Salamon¹

¹ University of St Andrews.

{ozgur.akgun, ian.gent, caj21, ijm, Andras.Salamon}@st-andrews.ac.uk

² University of York. {alan.frisch, peter.nightingale}@york.ac.uk

Abstract The ESSENCE language allows a user to specify a constraint problem at a level of abstraction above that at which constraint modelling decisions are made. ESSENCE specifications are refined into constraint models using the CONJURE automated modelling tool, which employs a suite of refinement rules. However, ESSENCE is a rich language in which there are many equivalent ways to specify a given problem. A user may therefore omit the use of domain attributes or abstract types, resulting in fewer refinement rules being applicable and therefore a reduced set of output models from which to select. This paper addresses the problem of recovering this information automatically to increase the robustness of the quality of the output constraint models in the face of variation in the input ESSENCE specification. We present reformulation rules that can change the type of a decision variable or add attributes that shrink its domain. We demonstrate the efficacy of this approach in terms of the quantity and quality of models CONJURE can produce from the transformed specification compared with the original.

1 Introduction and Background

The *modelling bottleneck* is the difficulty of formulating a problem of interest as a constraint model suitable for input to a constraint solver. The space of possible models for a given problem is typically large, and the model selected can have a dramatic effect on the efficiency of constraint solving. This presents a serious challenge for the inexperienced user, who has difficulty in formulating a good (or even correct) model, and motivates efforts to automate constraint modelling. In this paper we show that one source of difficulty to inexperienced users can be ameliorated by *type strengthening* rules.

In this paper our focus is on the refinement-based approach, where a user writes *abstract* constraint specifications that describe a problem above the level at which constraint modelling decisions are made. Abstract constraint specification languages, such as ESSENCE or Zinc, support abstract decision variables with types such as set, multiset, relation and function, as well as *nested* types, such as set of sets and multiset of relations. Problems can typically be specified very concisely in this way, as demonstrated by the examples in Figure 1.

However, existing constraint solvers do not support these abstract decision variables directly, so abstract constraint specifications must be *refined* into concrete constraint models.

Work on automation of aspects of constraint modelling can be grouped into distinct categories. Models can be learned from positive or negative examples [12,9,4], various kinds of queries [5,7,6], arguments [30], or from natural language descriptions [18]. It is possible to partially automate the transformation of medium-level solver-independent constraint models [29,24,31,23,25,26,27,21]. Closer to our work, implied constraints have been derived from a constraint model [16,11,10,8,20] and refinement of abstract constraint specifications has been considered [15] using the languages ESRA [13], ESSENCE [14], \mathcal{F} [17] and Zinc [22,19,28].

We here use ESSENCE [14] as our abstract problem specification language. ESSENCE supports abstract decision variables with *types* such as `set`, `mset` (denoting a multiset), `relation` and `function`, as well as *nested* types, such as `set of set` and `mset of relation`. Types are used to construct *domains*, which are abstract collections of objects of some common type but with possibly additional structure indicated by means of *domain attributes*. Problems can typically be specified very concisely in this way. Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. Our CONJURE system [3,1,2] employs refinement rules to convert an ESSENCE specification into the solver-independent constraint modelling language ESSENCE PRIME [29]. From ESSENCE PRIME we use SAVILE ROW [26] to translate the model into input for a particular constraint solver while performing solver-specific model optimisations.

ESSENCE is a rich language in which there are many equivalent ways to specify a given problem. It is possible, therefore, for a user to avoid the use of domain attributes or abstract types, resulting in fewer refinement rules being applicable and therefore a reduced set of output models from which to select. This paper addresses the problem of recovering this information automatically and hence increasing the robustness of the quality of the output constraint models. We present reformulation rules that can change the type of a decision variable or add attributes that shrink its domain. We demonstrate the efficacy of this approach in terms of the quantity and quality of models CONJURE can produce from the transformed specification compared with the original. All the methods described in this paper are implemented as part of the CONJURE system.³

2 Motivating Examples

ESSENCE is a rich language with a wide range of type constructors and domain attributes. CONJURE uses type constructors and domain attributes when selecting from its library of representations. CONJURE has special highly efficient

³ An archive containing several examples of robustness transformations performed by CONJURE can be found at the following repository:

<https://github.com/stacs-cp/ModRef2021-robustness>

representations for many specific domains. For example, sets of fixed size and total functions. These are dramatically more efficient during constraint solving than the representations for variable-size sets and partial functions. Hence it is absolutely vital that the type constructor and domain attributes are as specific as possible. In this section we will give some concrete examples of how this can be achieved.

In ESSENCE, a *domain attribute* (denoted in brackets after the name of a type in a domain) specialises the domain. For example, a function can be *surjective*, *injective* or *total* (ESSENCE functions are partial by default). The specification given in Figure 1a defines a partial surjective function. A partial surjective function from a set S to a set T where $|S| = |T|$ must necessarily be a total bijective function. Therefore we can strengthen this function to be function (bijective, total) Index --> Index. This has not changed the number of values in the domain, but CONJURE has specialised efficient representations for total functions, so this change can lead to dramatically better performance during constraint solving.

```
given n : int(1..)
letting Index be domain int(1..n)
find arrangement : function (surjective) Index --> Index
```

(a) An ESSENCE specification using function domains, before domain strengthening

```
given n : int(1..)
letting Index be domain int(1..n)
find arrangement : function (total, bijective) Index --> Index
```

(b) An ESSENCE specification using function domains, after domain strengthening

```
find x : relation of (int(1..3) * int(4..6) * int(7..9))
such that forall i : int(1..3) . forall k : int(7..9) . x(i,_,k) = {ε}
```

(c) Input ESSENCE specification with a relation decision variable

```
find x : function (total) (int(1..3), int(7..9)) --> int(4..6)
such that forall i : int(1..3) . forall k : int(7..9) . toRelation(x)(i,k,_) = {ε}
```

(d) Reformulating the above to a function domain and the toRelation operator

```
find x : function (total) (int(1..3), int(7..9)) --> int(4..6)
such that forall i : int(1..3) . forall k : int(7..9) . x(i,k) = ε
```

(e) Reformulating the above to remove the toRelation operator

Figure 1: ESSENCE domain and representation rule examples

The type of a domain can also make a dramatic difference to the quality of models produced by CONJURE. Our second class of reformulations identify when a type may be replaced by a more specific type (for example, replacing a multiset with a set).

The ESSENCE specification in Figure 1c contains a relation variable that would be better posed as a function. For each assignment to the first and third value in the relation, the relation is satisfied by exactly one assignment to the middle index. In this example, ϵ can be an arbitrary ESSENCE expression possibly involving other decision variables. The second index is functionally defined by the first and third indices, so we can replace the relation with a function as shown in Figure 1d. To simplify the process of changing the relation to a function, we have replaced all occurrences of the relation x in the constraints with `toRelation(x)`. The `toRelation` operator maps the function back into a relation, and allows us to be sure we can transform any constraint involving the relation x into a constraint on the function x . We also reordered the indices of the relation to place the functionally defined index (or indices) at the end.

Figure 1d has reduced the domain size of the variable in our specification from $2^{3^3} = 134,217,728$ to $3^9 = 19,683$. Reducing the size of a domain strongly indicates (but does not guarantee) that the specification has been improved, because we have a much smaller domain to represent and search in our constraint solver. The final step in the reformulation is removing the `toRelation` operators where possible. This results in the final specification given in Figure 1e.

3 Domain Strengthening via Attribute Recovery

Refinement rules in ESSENCE rewrite an abstract structure into a more concrete equivalent. Domain attributes provide additional information that can be used for more effective refinement. This information can also be provided in an ESSENCE specification in the form of constraints. However, CONJURE uses only the domain and attributes of a variable to select its representation, so using constraints instead of domain attributes reduces the quantity and quality of models output by CONJURE. In this section we describe how an ESSENCE specification can be reformulated to recover latent attribute information that was omitted from the specification. There exists a strengthening rule to recover each attribute in ESSENCE, as we will demonstrate below.

A *strengthening rule* takes as input a domain and optionally a constraint expression and outputs a new domain. In addition, it indicates whether the input constraint should remain (by default it is removed). For some rules the entire constraint is subsumed by the addition of the attribute so the constraint is not required, in others the constraint is still required. The strengthening rules are applied by CONJURE before the representation of each variable is chosen. The strengthening rules presented in this paper take a similar form to the CONJURE rules presented previously [3]. Each rule takes as input a single decision variable and one or more constraints, and outputs a new modified variable and one or more constraints. Each rule removes the constraints matched by the input of the

rule unless we state otherwise. All other constraints involving the variable are unchanged, except referring to the new name for the variable.

CONJURE rules contain meta-variables to denote expressions which must be matched, such as the $\&n$ in Figure 3a. These meta-variables can match an arbitrary ESSENCE expression, not just a single identifier. This means our rules can match large ESSENCE expressions with simple patterns. Because the identifier $\&n$ in Figure 3a is also used in the `size` attribute of a `set`, it must be a constant or parameter, not a decision variable. The rule matcher will automatically reject any $\&n$ which does not meet this requirement. Other identifiers, such as $\&exp$ in Figure 4a, will match with any ESSENCE expression at all, including expressions involving decision variables.

General CONJURE rewrite rules (such as a rule which normalises $A > B$ to $B < A$) can match partial expressions contained within constraints. Strengthening rules must only be applied to top-level constraints. In ESSENCE abstract domain constructors can be nested arbitrarily, and attributes can be recovered for domains nested inside other domain constructors. This is done by CONJURE for all rules automatically, and does not require writers of rules to worry about deeply nested types. The example given in Figure 2a shows an example where we place a constraint on all the `mset` members of `S`. CONJURE automatically recognises the `forall m in S` quantifier imposes the constraint on all members of `S`, and so the rule in Figure 3b which is designed for `mset` variables will trigger, adding to the members of `S` the attribute `(maxOccur 3)`, producing the output in Figure 2b.

3.1 Recovering Size and Occurrence Related Attributes

In ESSENCE abstract domain constructors — `set`, `mset`, `function`, `relation`, and `partition` — can have `size`, `minSize`, and `maxSize` attributes. CONJURE can recover these attributes when a cardinality constraint is posted at the top level in the problem specification. For example, a constraint of the form $|x| = n$ implies the recovery of a `size` attribute for `x`; the constraints $|x| < n$ and $|x| \leq n$ imply a `maxSize` attribute with the values $n-1$ and n respectively. The recovery of the `minSize` attribute is handled similarly.

In addition to the three *size* attributes common to all abstract domain constructors, `mset` and `partition` have additional attributes which are handled similarly. For `mset`, the attributes `minOccur` and `maxOccur` constrain the number of occurrences of individual values. For `partition` the attributes `partSize`, `minPartSize`, `maxPartSize` constrain the sizes of the parts in the partition, and `numParts`, `minNumParts` constrain the number.

Figure 3a presents a recovery of the `size` attribute for sets. This rule as given here is specialised to just the `set` type constructor, however in CONJURE this rule is implemented to handle any abstract domain constructor. Figure 3b presents a recovery of the `maxOccur` attribute for `mset`. This rule, as well as all other strengthening rules also work when the `mset` domain is nested inside other abstract domain constructors. Figure 2a presents an example of a nested type where we recover attributes for both the inner and outer type constructors.

```

find S : set of mset of int(0..9)
such that
  |S| = 2,
  forall m in S . forall i : int(0..9) . freq(m,i) <= 3

```

(a) The input ESSENCE problem specification

```

find S: set (size 2) of mset (maxOccur 3) of int(0..9)

```

(b) Recovered attributes

Figure 2: An example ESSENCE specification with nested domains.

Input	find &x : set of &T
Input	&x = &n
Output	find &x : set (size &n) of &T

(a) Recovering the size attribute for sets

Input	find &x : mset of &T
Input	forall &i : &T . freq (&x,&i) <= &n
Output	find &x : mset (maxOccur &n) of &T

(b) Recovering the maxOccur attribute for multi-sets

Figure 3: Rules for the domains in Figure 2.

In Figure 2a, the domain given of s is infinite. While CONJURE requires all variables have a finite domain, the check for finiteness is performed after type-strengthening. This shows another way in which type strengthening can help users model their problems more easily.

3.2 Recovering Special function and sequence Attributes

In addition to the common `minSize`, `maxSize` and `size` attributes, functions have four additional attributes: `total`, `injective`, `surjective`, and `bijective`. Figure 4a gives a strengthening rule, where the `total` attribute is inferred if there is a constraint to assign values to all mappings in the function. Here, unlike most other strengthening rules, the constraint is not removed because it contains more information than just representing a `total` attribute. Figure 4b gives a strengthening rule, where the `surjective` attribute is inferred if there is a constraint stating for all values in the range of the function there is a mapping. Similarly, Figure 4c gives a strengthening rule where the `injective` attribute is inferred if there is a constraint stating the image of the function to be distinct for distinct values. In both of these rules, the constraints are removed from the model because they are subsumed by adding the suitable attribute to x . CON-

Input	<code>find &x : function &T_1 --> &T_2</code>
Input	<code>forall &i : &T_1 . &x(&i) = &exp</code>
Output	<code>find &x : function (total) &T_1 --> &T_2</code> <i>The constraint remains unchanged.</i>

(a) Recovering the `total` attribute for functions

Input	<code>find &x : function &T_1 --> &T_2</code>
Input	<code>forall &j : &T_2 . exists &i : &T_1 . f(&i) = &j</code>
Output	<code>find &x : function (surjective) &T_1 --> &T_2</code>

(b) Recovering the `surjective` attribute for functions

Input	<code>find &x : function &T_1 --> &T_2</code>
Input	<code>forall &i, &j : &T_1 . &i != &j -> &x(&i) != &x(&j)</code>
Output	<code>find &x : function (injective) &T_1 --> &T_2</code>

(c) Recovering the `injective` attribute for functions

Figure 4: Strengthening rules for attributes of function domains

JURE will further infer the `bijective` attribute for any variable with both the `injective` and `surjective` attributes.

Sequence domains support the `injective`, `surjective` and `bijective` attributes as well. These are handled similarly to functions.

3.3 Recovering Special `relation` Attributes

Figure 5 gives strengthening rules to infer `functional` and `total_functional` attributes. These two attributes restrict the domain of a relation so some columns of a relation are functionally determined by the rest of the columns. `functional` restricts the functionally determined columns take at most one assignment for each assignment to the other columns, `total_functional` restricts the functionally defined columns to take exactly one assignment. For example, a binary relation `r` together with the constraint `forall i : dom . |r(i,_)| = 1` can be turned into a function mapping values from the first column to the second one. In addition, such a function domain has to be `total`, because there is exactly one value of the second column for each value of the first. The constraint `forall i : dom . |r(i,_)| <= 1` would let CONJURE recover a `functional` attribute instead of `total_functional`.

3.4 Recovering Special `partition` Attributes

Figure 6 gives strengthening rules for the `partition` type constructor. In the first the `regular` attribute is inferred if there is a constraint forcing all parts in the partition to have the same cardinality. As discussed earlier in Section 3.1, CONJURE also recovers the `numParts` and `partSize` attributes for `partition`.

Input	find $\&x$: relation of ($\&T_1 * \&T_2 * \&T_3$)
Input	$ \&x(\&a, \&b, _) \leq 1$
Output	find $\&x$: relation (functional (1,2)) of ($\&T_1 * \&T_2 * \&T_3$)

Input	find $\&x$: relation of ($\&T_1 * \&T_2 * \&T_3$)
Input	$ \&x(\&a, \&b, _) = 1$
Output	find $\&x$: relation (total_functional (1,2)) of ($\&T_1 * \&T_2 * \&T_3$)

Input	find $\&x$: relation of ($\&T_1 * \&T_2 * \&T_3$)
Input	$\&x(\&a, \&b, _) = \&c$
Output	find $\&x$: relation (total_functional (1,2)) of ($\&T_1 * \&T_2 * \&T_3$) <i>The constraint remains unchanged.</i>

Figure 5: Strengthening rules for attributes of relation domains

Input	find $\&x$: partition from $\&T$
Input	forall $\&i, \&j$ in $\text{parts}(\&x)$. $ \&i = \&j $
Output	find $\&x$: partition (regular) from $\&T$

Figure 6: Strengthening rules for attributes of partition domains

3.5 Domain-Only Recovery

It is sometimes possible to recover domain attributes just from the domain of a variable. For example in Figure 1a, a surjective function between two domains of equal size must be total and bijective. CONJURE contains a range of similar rules for other types. For example for a mset domain `maxSize n` implies `maxOccur n` and `minOccur n` implies `minSize n`. Further, for partitions if the `partSize` multiplied by the `numParts` is equal to the size of the set the partition is defined over, the partition is complete and regular. While these modifications do not remove any constraints, they allow CONJURE to choose from a wider range of representations, as representation selection is based upon the domain and attributes.

3.6 Limitations of Attribute Recovery

Each of CONJURE's attribute recovery rules match an explicit pattern of constraints and attributes. The reformulation rules in CONJURE can reduce other constraints to fit into these patterns, but there are occasions where we fail to detect attributes. Theorem 1 shows this is inevitable, as detecting if we can add an attribute to a variable is equivalent to the halting problem.

Theorem 1. *For any ESSENCE attribute A , suppose there is an oracle that can decide, for any ESSENCE specification S containing variable V , whether V can have the attribute A attached to it without removing solutions. Then this oracle*

can be used to solve the halting problem, unless A is satisfied by every variable it can be attached to.

Proof. To reduce from the Halting problem, consider determining if a Turing machine T , with a distinguished state q and no halting states, ever reaches q when started on a blank tape. This problem is Σ_0^1 -complete. We construct an ESSENCE specification which takes a parameter n and has a solution if T reaches q within n steps, since T cannot reach any part of the tape that is more than n steps from the starting position. Detecting if this ESSENCE specification has a solution for any n is therefore equivalent to solving the halting problem. Assuming that A is not satisfied by some variable V , we now add V to this ESSENCE specification, placing no constraints on the variable V . If T never reaches state q , the problem has no solutions and so we can add any extra attribute to V without affecting the set of solutions. If T does reach state q after some number of steps n , then the set of solutions includes (possibly many copies of) all assignments to V , so the addition of A will reduce the set of solutions and is invalid. The supposed oracle therefore solves the halting problem.

4 Type Strengthening

As well as adding attributes to domains, CONJURE also implements *type strengthening* rules which transform one domain into another, e.g. turning a `relation` into a `function` or `mset` into `set`. This provides a greater set of representational choices.

4.1 Overview

Our implementation of type strengthening builds directly upon domain strengthening. We split type strengthening into two parts. Firstly, for each type \mathbb{T} which can be strengthened to another type \mathbb{U} , there is an attribute on \mathbb{T} which restricts a variable of type \mathbb{T} to assignments of type \mathbb{U} , and every other attribute on \mathbb{T} is also valid on \mathbb{U} . This means detecting type strengthening uses only the domain of a variable. Further, for each type \mathbb{T} which can be strengthened into another type \mathbb{U} , we have an operator toT which transforms a \mathbb{U} back into a \mathbb{T} . This allows us to replace a variable $\mathbb{T} \ t$ with a variable $\mathbb{U} \ u$, replacing all occurrences of t with $\text{toT}(u)$. We then use CONJURE's standard rewrite engine to simplify the resulting specification.

4.2 MSet to Set

The rule below demonstrates transforming a `mset` with `(maxOccur 1)` into a `set`. When applying this rule every occurrence of `&x` in the model is replaced with `toMSet(&x)`. For example, `freq(&x, 2) = 0` turns into `freq(toMSet(&x), 2) = 0`. Later CONJURE simplifies this expression to `!(2 in &x)` by applying one of its rewrite rules.

Input	<code>find &x : mset (maxOccur 1) of &T</code>
Output	<code>find &x : set of &T</code>

4.3 Relation to Function

We previously discussed in Section 3.3 how we can detect relations which are functional in one or more of their indices. Such relations can be transformed into functions. In general these transformations involve arbitrary arity relations with an arbitrary subset of the indices of the relation defining the domain of the function. In this section we will provide some concrete examples for arity 3 relations.

The two rules below show examples of transforming a relation to a partial and total function. When transforming a relation into a function, we replace all occurrences of the relation x with the expression `toRelation(x)`. This leads to specifications like Figure 1d, where we use `toRelation` and project the resulting relation. CONJURE simplifies this example to the specification given in Figure 1e.

Input	<code>find &x : relation (functional (1,2)) of (&T_1 * &T_2 * &T_3)</code>
Output	<code>find &x : function (&T_1 * &T_2) --> &T_3</code>

Input	<code>find &x : relation (total_functional (1,3)) of (&T_1 * &T_2 * &T_3)</code>
Output	<code>find &x : function (total) (&T_1 * &T_3) --> &T_2</code>

5 Conclusions

We have shown how we can make the powerful and expressive type system of ESSENCE more robust and easier to use. Type and domain strengthening make ESSENCE more useful for beginners who have not yet learnt, or do not wish to learn, the full list of attributes available in ESSENCE. Furthermore, automated type and domain strengthening allows for new attributes to be added to the ESSENCE language which can improve the performance of an ESSENCE specification without the user having to make any changes to their model.

In future work, we plan to extend type and domain strengthening to more general kinds of rewriting of ESSENCE specifications. For instance, currently we cannot recover the injectivity of f and g from

<code>forall i, j . f[i] = j <-> g[j] = i</code>
--

via our existing type and domain strengthening rules.

Acknowledgements This research was supported by the UK EPSRC grants EP/K015745/1 and EP/V027182/1. Chris Jefferson is a University Research Fellow funded by the Royal Society.

References

1. Akgun, O., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kothoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: CP. pp. 107–116. LNCS 8124 (2013). https://doi.org/10.1007/978-3-642-40627-0_11
2. Akgun, O., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI. pp. 3–8 (2014). <https://doi.org/10.3233/978-1-61499-419-0-3>
3. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAI. pp. 4–11 (2011), <https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3687>
4. Arcangioli, R., Bessiere, C., Lazaar, N.: Multiple constraint acquisition. In: IJCAI. pp. 698–704 (2016), <https://www.ijcai.org/Abstract/16/105>
5. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: CP. pp. 141–157. LNCS 7514 (2012). https://doi.org/10.1007/978-3-642-33558-7_13
6. Bessiere, C., Coletta, R., Daoudi, A., Lazaar, N.: Boosting constraint acquisition via generalization queries. In: ECAI. pp. 99–104 (2014). <https://doi.org/10.3233/978-1-61499-419-0-99>
7. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C.G., Walsh, T.: Constraint acquisition via partial queries. In: IJCAI. p. 7 (2013), <https://www.ijcai.org/Abstract/13/078>
8. Bessiere, C., Coletta, R., Petit, T.: Learning implied global constraints. In: IJCAI. pp. 44–49 (2007), <https://www.ijcai.org/Proceedings/07/Papers/005.pdf>
9. Bessiere, C., Koriche, F., Lazaar, N., O’Sullivan, B.: Constraint acquisition. *Artificial Intelligence* **244**, 315 – 342 (2017). <https://doi.org/10.1016/j.artint.2015.08.001>
10. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: ECAI. pp. 73–77 (2006), <https://ebooks.iospress.nl/volumearticle/2653>
11. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: CP. pp. 575–579. LNCS 2239 (2001). https://doi.org/10.1007/3-540-45578-7_42
12. De Raedt, L., Passerini, A., Teso, S.: Learning constraints from examples. In: AAI. pp. 7965–7970 (2018), <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/viewPaper/17229>
13. Flener, P., Pearson, J., Ágren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: LOPSTR 2003. LNCS 3018 (2004). https://doi.org/10.1007/978-3-540-25938-1_18
14. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. *Constraints* **13**(3), 268–306 (2008). <https://doi.org/10.1007/s10601-008-9047-y>
15. Frisch, A.M., Jefferson, C., Martínez-Hernández, B., Miguel, I.: The rules of constraint modelling. In: IJCAI. pp. 109–116 (2005), <http://www.ijcai.org/papers/1667.pdf>
16. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: A system for transforming constraint satisfaction problems. In: Recent Advances in Constraints. pp. 15–30. LNCS 2627 (2003). https://doi.org/10.1007/3-540-36607-5_2

17. Hnich, B.: Thesis: Function variables for constraint programming. *AI Communications* **16**(2), 131–132 (2003), <https://content.iospress.com/articles/ai-communications/aic281>
18. Kiziltan, Z., Lippi, M., Torroni, P.: Constraint detection in natural language problem descriptions. In: *IJCAI*. pp. 744–750 (2016), <https://www.ijcai.org/Abstract/16/111>
19. De Koninck, L., Brand, S., Stuckey, P.J.: Data independent type reduction for Zinc. In: *ModRef* (2010), https://people.eng.unimelb.edu.au/pstuckey/papers/type_reduction.pdf
20. Leo, K., Mears, C., Tack, G., De La Banda, M.G.: Globalizing constraint models. In: *CP*. pp. 432–447. LNCS 8124 (2013). https://doi.org/978-3-642-40627-0_34
21. Little, J., Gebruers, C., Bridge, D., Freuder, E.C.: Using Case-Based reasoning to write constraint programs. In: *CP*. p. 983. LNCS 2833 (2003). https://doi.org/978-3-540-45193-8_107
22. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. *Constraints* **13**(3), 229–267 (2008). <https://doi.org/10.1007/s10601-008-9041-4>
23. Mills, P., Tsang, E.P.K., Williams, R., Ford, J., Borrett, J.: *EaCL 1.5: An easy abstract constraint optimisation programming language*. Tech. Rep. CSM 324, University of Essex, Colchester, UK (Feb 1999), <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.170.568>
24. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: *MiniZinc: Towards a standard CP modelling language*. In: *CP*. pp. 529–543. LNCS 4741 (2007). https://doi.org/10.1007/978-3-540-74970-7_38
25. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: *CP*. pp. 590–605. LNCS 8656 (2014). https://doi.org/10.1007/978-3-319-10428-7_43
26. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. *Artificial Intelligence* **251**, 35–61 (2017)
27. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: *CP*. pp. 330–340. LNCS 9255 (2015). https://doi.org/10.1007/978-3-319-23219-5_23
28. Rafeh, R., Jaber, N.: *LinZinc: A library for linearizing Zinc models*. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering* **40**(1), 63–73 (2016). <https://doi.org/10.1007/s40998-016-0005-1>
29. Rendl, A.: *Effective compilation of constraint models*. Ph.D. thesis, University of St Andrews (2010), <http://hdl.handle.net/10023/973>
30. Shchekotykhin, K., Friedrich, G.: Argumentation based constraint acquisition. In: *Int. Conf. on Data Mining*. pp. 476–482 (2009). <https://doi.org/10.1109/ICDM.2009.62>
31. Van Hentenryck, P.: *The OPL Optimization Programming Language*. MIT Press (1999)