Towards Exploratory Reformulation of Constraint Models

Ian Miguel ✉ 🆔
School of Computer Science, University of St Andrews, UK

András Z. Salamon ✉ 🆔
School of Computer Science, University of St Andrews, UK

Christopher Stone ✉ 🆔
School of Computer Science, University of St Andrews, UK

―――― **Abstract** ――

It is well established that formulating an effective constraint model of a problem of interest is crucial to the efficiency with which it can subsequently be solved. Following from the observation that it is difficult, if not impossible, to know *a priori* which of a set of candidate models will perform best in practice, we envisage a system that explores the space of models through a process of reformulation from an initial model, guided by performance on a set of training instances from the problem class under consideration. We plan to situate this system in a refinement-based approach, where a user writes a constraint specification describing a problem above the level of abstraction at which many modelling decisions are made. In this position paper we set out our plan for an exploratory reformulation system, and discuss progress made so far.

1 Introduction

It is well established that formulating an effective constraint model of a problem of interest is crucial to the efficiency with which it can subsequently be solved [19]. This has motivated a variety of approaches to automating the modelling process. Some learn models from, variously, natural language [27], positive or negative examples [17, 11, 5], membership queries, equivalence queries, partial queries [7, 9], generalisation queries [8] or arguments [43]. Other approaches include: automated transformation of medium-level solver-independent constraint models [41, 34, 45, 32, 37, 36, 38]; deriving implied constraints from a constraint model [22, 16, 15, 10, 29]; case-based reasoning [30]; and refinement of abstract constraint specifications [21] in languages such as ESRA [18], Essence [20], $\mathcal{F}$ [25] or Zinc [31, 28, 40].

Following from the observation that it is difficult, if not impossible, to know *a priori* which of a set of candidate models will perform best in practice, we envisage a system that *explores* the space of models through a process of reformulation from an initial model, guided by performance on a set of training instances from the problem class under consideration.

We plan to situate this system in a refinement-based approach, where a user writes a constraint specification describing a problem above the level of abstraction at which many modelling decisions are made. The advantage of proceeding from a problem specification rather than a concrete constraint model are that the structure apparent in a concise abstract specification, which may be obscured in concrete model, can help to guide reformulation. Furthermore, a single reformulated specification can be refined into a variety of both models and solving paradigms, allowing us to gain a fuller picture of performance.

In the remainder of this position paper we set out our plan for an exploratory reformulation system, and discuss progress made so far.

2   Background: Essence and the Constraint Modelling Pipeline

The refinement-based approach in which we intend to implement exploratory reformulation is the constraint modelling pipeline that takes an abstract problem specification in Essence [20] as its input. Essence is a well-established declarative language for constraint programming, supported by the Athanor local search solver [6] and the Conjure [3] and Savile Row [36] translators working in concert with many back-end solvers such as the SAT solvers Cadical and Kissat [12] or the constraint solver Minion [23]. In this section we give a brief overview of the process of producing constraint models from Essence input.

An illustrative Essence specification of the Progressive Party Problem (problem 13 at CSPLib) is presented in Figure 1. The natural language description of the problem, taken from CSPLib [26], is:

> *The problem is to timetable a party at a yacht club. Certain boats are to be designated hosts, and the crews of the remaining boats in turn visit the host boats for several successive half-hour periods. The crew of a host boat remains on board to act as hosts while the crew of a guest boat together visits several hosts. Every boat can only hold a limited number of people at a time (its capacity) and crew sizes are different. The total number of people aboard a boat, including the host crew and guest crews, must not exceed the capacity. A guest boat cannot not revisit a host and guest crews cannot meet more than once. The problem facing the rally organizer is that of minimizing the number of host boats.*

An Essence specification identifies: the input parameters of the problem class (`given`), whose values define an instance; optional further constraints on allowed parameter values (`where`); the combinatorial objects to be found (`find`); and the constraints the objects must satisfy (`such that`). An objective function may be specified (`minimising` in the example) and identifiers declared (`letting`). Essence supports a number of abstract type constructors, such as relations, functions, sequences, sets, and partitions. These may be arbitrarily nested, such as the set of functions that represents the schedule in the example.

The abstract decision variables supported by Essence are not typically supported directly by solvers, and so an Essence specification must be *refined* via the automated modelling tool Conjure [3] into the generic constraint modelling language Essence Prime [35]. There are generally many different refinement pathways, depending on decisions as to how to represent the decision variables and the constraints on them, whether and how to break symmetry, whether to channel between different representations, and so on, each leading to a different constraint model. Conjure features various heuristics so as to select a good model automatically. The Essence Prime model is then prepared for input to a particular solver by Savile Row [36]. Depending on the target, e.g. SAT vs CP vs SMT, further modelling decisions are required at this stage.

3   Exploratory Reformulation of Essence Specifications

Even though both Conjure and Savile Row feature heuristics to refine a high quality model for a target solver, the refinement process is heavily influenced by the Essence

```
given n_boats, n_periods : int(1..)
letting Boat be domain int(1..n_boats)
given capacity, crew : function (total) Boat --> int(1..)

find hosts : set of Boat,
     sched : set (size n_periods) of function (total) Boat --> Boat
minimising |hosts|

such that
   $ Hosts remain the same throughout the schedule
   forAll p in sched . range(p) subsetEq hosts,
   $ Hosts stay on their own boat
   forAll p in sched . forAll h in hosts . p(h) = h,
   $ Hosts have the capacity to support the visiting crews
   forAll p in sched . forAll h in hosts .
      (sum b in preImage(p,h) . crew(b)) <= capacity(h),
   $ No two crews are at the same party more than once
   forAll b1,b2 : Boat . b1 < b2 ->
      (sum p in sched . toInt(p(b1) = p(b2))) <= 1
```

◼ **Figure 1** Essence Specification of the Progressive Party Problem.

specification from which refinement proceeds. By reformulating the specification we can open up new refinement possibilities and therefore new models.

The reformulations we envisage include the transformation of the logical and arithmetic expressions in the specification, which will affect how it is refined to a constraint model and encoded for a solver. Furthermore, by choosing to reformulate at the Essence level rather than a constraint model, we can take advantage of the structural information present in the abstract types that Essence provides.

To illustrate, we present a simple example reformulation of the Progressive Party Problem specification. The decision variable `sched` is a fixed-cardinality (for the number of periods) set of total functions. For each such function we might consider if we can further constrain its domain and range. Since the functions are total their domain is fixed to the set of boats. The range of each function has size at least one, since all boats have an image, and at most `n_boats` if these images are distinct.

The constraint:

```
$ Hosts remain the same throughout the schedule
forAll p in sched . range(p) subsetEq hosts,
```

connects the size of the range of each function to that of the hosts set. Since `range(p)` is a subset of `hosts` and we showed above that `range(p)` has size at least one, `hosts` cannot be empty, so we can strengthen the `find` statement:

```
find hosts : set (minSize 1) of Boat
```

From the above and the constraint:

```
forAll p in sched . forAll h in hosts . p(h) = h
```

we can prove that `range`(p) = hosts, strengthening the first of the constraints in the specification: since `hosts` is a set, the `h` in the image of the function are distinct. So, `forAll h in hosts . p(h) = h` tells us that the range is at least the size of `hosts`. We showed that the range is a subset of the hosts, and it is the same size, so they are equal.

We could go further still to realise that each total function is in fact partitioning the boats into `|hosts|` parts, and reformulate the decision variable as follows:

```
find sched : set (size n_periods) of partition from Boat
```

In general, for any formulas $a(x)$ and $b(x)$ in which the variable x appears free, from the constraint `forall h in hosts . (a(h) < b(h))` we can derive the implied constraint `(sum h in hosts . a(h)) < (sum h in hosts . b(h))`. Hence, from:

```
$ Hosts have the capacity to support the visiting crews
forAll p in sched . forAll h in hosts .
   (sum b in preImage(p,h) . crew(b)) <= capacity(h),
```

we can derive that the sum of crew sizes is less than or equal to sum of host capacities:

```
(sum h in hosts . (sum b in preImage(p,h) . crew(b)))
   < (sum h in hosts . capacity(h))
```

This collection of relatively simple reformulations is indicative of those we intend to build into our system. The hope is that combinations of simple reformulations can, in aggregate, make a significant improvement to an input specification. Of course, it is difficult to know whether a particular reformulation, or sequence of reformulations, will have a positive effect. In the above example, would we expect the total function or partition specification to perform better? The answer is that it depends exactly how these abstract structures are refined and encoded.

This motivates the need for exploratory reformulation, in which we consider the performance of different sequences of reformulations on a set of training instances for the problem class studied. Our proposal can also be seen as an extension of [4], which described heuristics to guide the choice of rewrite rules to apply to an ESSENCE specification. The focus in that work was *type strengthening*, whereby properties expressed by constraints can sometimes be expressed instead by additional type information, allowing more effective model refinement.

In a more exploratory setting, it is unlikely that an exhaustive search through all possible reformulations will be possible. In order to control the exploration of new reformulation sequences versus the exploitation of existing sequences by extending them, we plan to employ Monte Carlo Tree Search, as has recently proven successful in the generation of streamliner constraints [44]. Given a resource budget, the system will then explore a number of promising reformulated specifications in an attempt to improve on the original. The cost of this process is then amortised over the remainder of the problem class.

This process is illustrated in Figure 2. The upper part of the tree of possible reformulations is maintained explicitly. Every iteration begins with a selection phase, which uses a policy such as Upper Confidence Bound applied to Trees (UCT) [13] to traverse the explored part of the tree until an unexpanded node is reached. The selected node is then expanded by randomly selecting a child, i.e. a reformulation applicable to the specification represented by the selected node. The new reformulated specification is then evaluated against a set of training instances and the results back-propagated up the tree to influence the selection of the next node to expand.

In the remainder of this paper, we discuss our progress to date in implementing an exploratory reformulation framework for ESSENCE specifications.
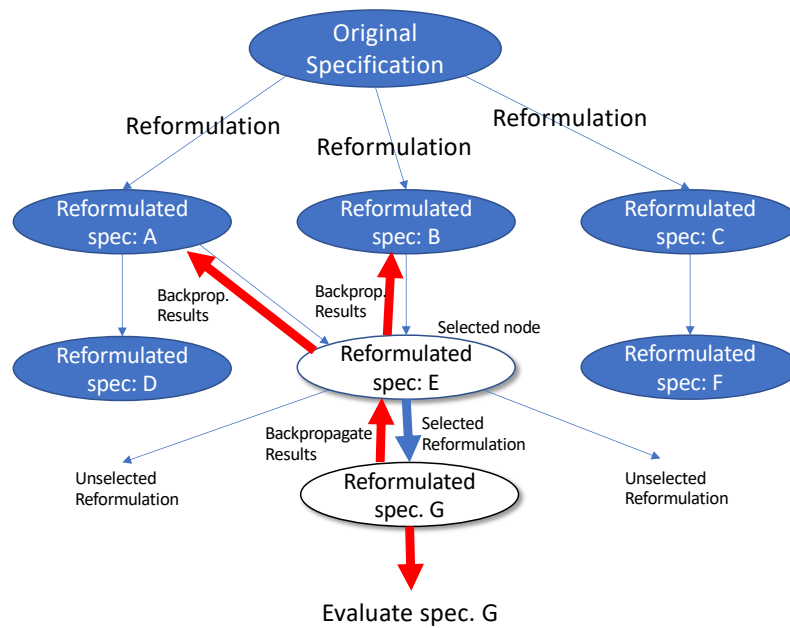
Figure 2 Exploratory Reformulation via Monte Carlo Tree Search.

4 Current Progress

4.1 Reformulation, redux

Reformulation can be conceptualised in many different ways. Here we have chosen to pursue one particular framework for reformulation, based on rewriting.

The specification at each stage of reformulation can be represented as an abstract syntax tree (AST). A specification can be recovered from the corresponding AST without loss of information. ASTs are often modelled as trees. It is also possible to replace common subtrees in the AST by pointers which turns the data structure into a form of directed acyclic graph. Either way, ASTs can be represented as graphs consisting of a set of labelled vertices together with a set of directed and labelled arcs between vertices. A *node* in the AST is a labelled vertex.

A graph rewriting system nondeterministically matches a pattern graph to the target graph; if a match is found then the part of the target graph that is matched is replaced according to the rewrite rule by a different graph. The details of matching and the kinds of rewriting can vary, but the graph rewriting paradigm is general enough to be Turing-complete [24] and thus can capture the reformulation sequences that we want to study.

Each reformulation in a sequence of reformulations can be treated as a step taken by a graph rewriting system acting on the AST as the target graph being rewritten. Each kind of reformulation is expressed by a graph rewrite rule.

We have used the graph rewriting language GP2 [39] to perform rewriting on the abstract syntax tree representing a specification. The GP2 system includes a flexible language in which to express graph rewriting rules, with an efficient implementation of the rewriting engine optimised for sparse large graphs [14].

4.2   System development

Since ESSENCE is a language with many features, we have initially defined a subset of the ESSENCE language (Emini) that is sufficiently expressive to capture the full power of ESSENCE itself, but with less syntax. In particular, an Emini specification is also valid ESSENCE. Emini therefore inherits the decidability of satisfiability of ESSENCE specifications. The Emini language uses tuples and relations, integer and Boolean types, and allows most ESSENCE expressions (including quantification, inequalities, Boolean logic, and arithmetic). This choice of types was guided by previous work on the expressivity of ESSENCE [33], which showed that nested relation types are sufficient to express all problems in the polynomial hierarchy. Over time, we intend to extend our system to the entire ESSENCE language.

We have implemented our system in Python. An AST representation of Emini is our core data structure. The AST is represented as nested Python objects and labels which are produced by a parser developed for Emini. The AST can be translated to and from several other different formats. Translations implemented to date are illustrated in Figure 3.

Among these alternative formats, the NetworkX AST representation provides access to a variety of tools, such as plotting, graph algorithms, and machine learning libraries. The JSON format allows easy data interchange, and the GP2 format allows the application of graph rewriting rules to our specifications.

One application of our system is pretty printing. Reading in an Emini specification, and then writing out an Emini representation of the AST, yields a specification in a normalised format with superfluous parentheses and syntactic sugar removed. Optionally, grammatical information about each node can be printed as in the following simple example specification:

■ **Listing 1** Example specification

```
find x : int(0..100)
such that
    1*(2+3)*4 = x
```

where we can visualise the AST of this specification as:

■ **Listing 2** AST

```
└ root  #Node
   ├ find  #FindStatement
   │ └ x  #DecisionVariable
   │    └ int  #IntDomain
   │       ├ 0  #Integer
   │       └ 100  #Integer
   └ such that  #SuchThatStatement
      └ =  #BinaryExpression
         ├ *  #BinaryExpression
         │ ├ *  #BinaryExpression
         │ │ ├ 1  #Integer
         │ │ └ +  #BinaryExpression
         │ │    ├ 2  #Integer
         │ │    └ 3  #Integer
         │ └ 4  #Integer
         └ x  #ReferenceToDecisionVariable
```
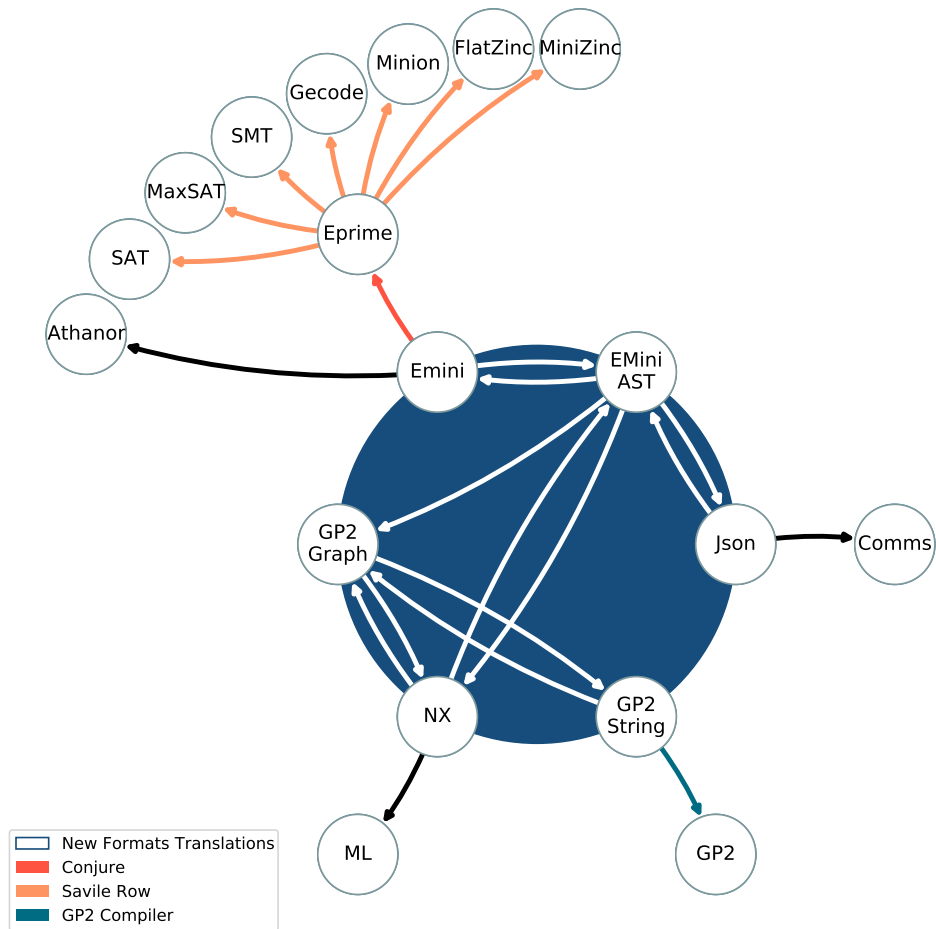
Figure 3 Mapping all formats and transformations. The white arrows are all the novel translations that have been implemented. Abbreviations: ML=Machine learning, Comms=Communication, NX=NetworkX graph.
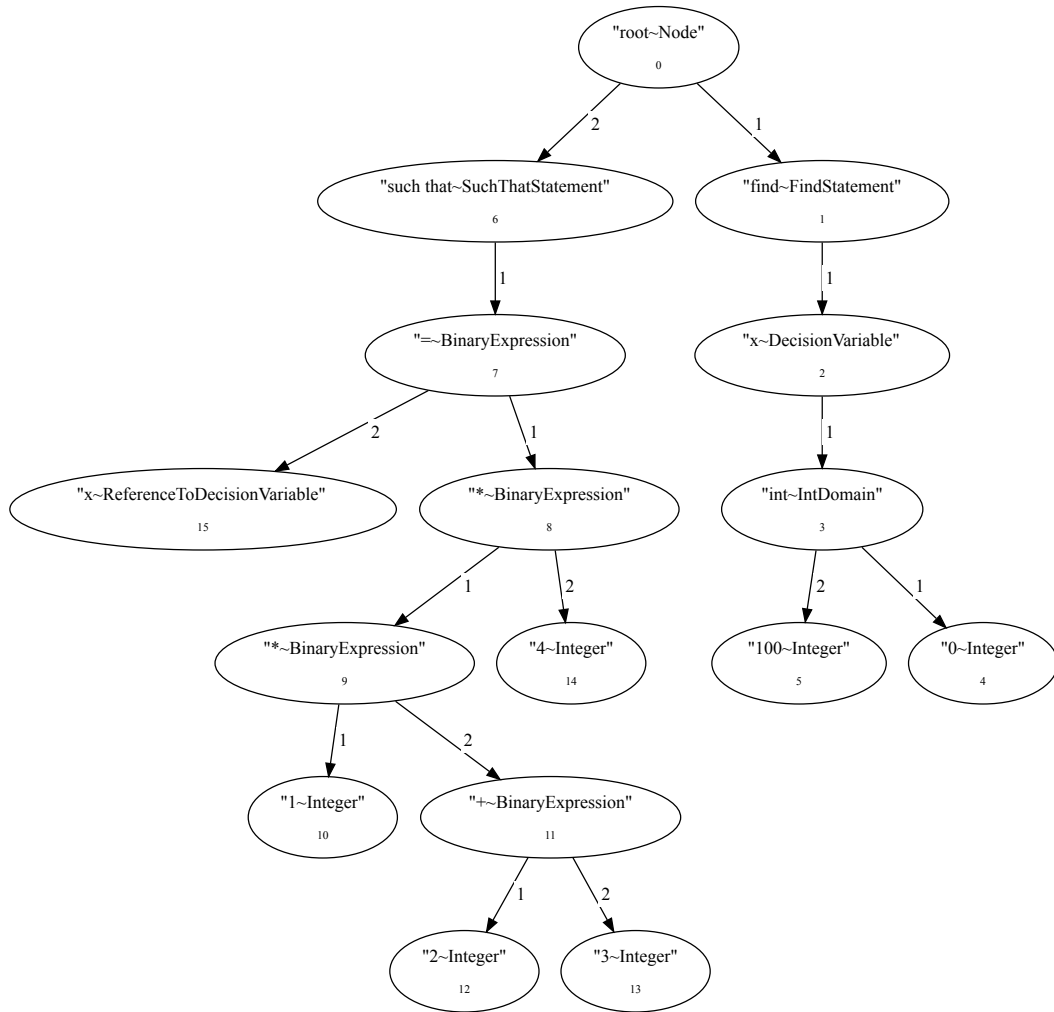
■ **Figure 4** Visualisation of the GP2 graph representation.

We implement our rewrite rules as GP2 programs, and use the GP2 graph rewriting system to perform the rewriting nondeterministically. The advantage of using GP2 is that this rewriting system performs well even on large graphs, and many different rewrite rules can be applied at once. In the GP2 language graphs are specified by a list of vertices, represented by tuples (index, label) and a list of edges represented by tuples (index, source, target, label). We store grammatical information in each node, and the parent-child relations become edges. The ordering of a parent's children is represented by positive integers in the edge label, with 1 denoting the first child. The simple specification Listing 1, with the AST in Listing 2, is depicted in Figure 4 as a GP2 graph.

The fundamental components of GP2 programs are rewrite rules. Each rule is expressed as a pair of graphs that determine the precondition and postcondition of a matched subgraph. Figure 5 shows a simple example of a rule interchanging the operands of a commutative operation. Albeit trivial, this rewrite rule can already be used to test the behaviour of solvers over different variable orderings, and if equipped with an additional *where* statement and a comparison, it could be used to normalise specifications.

We are currently investigating appropriate rewrite rules.

commute(operator, operand1, operand2 : string)
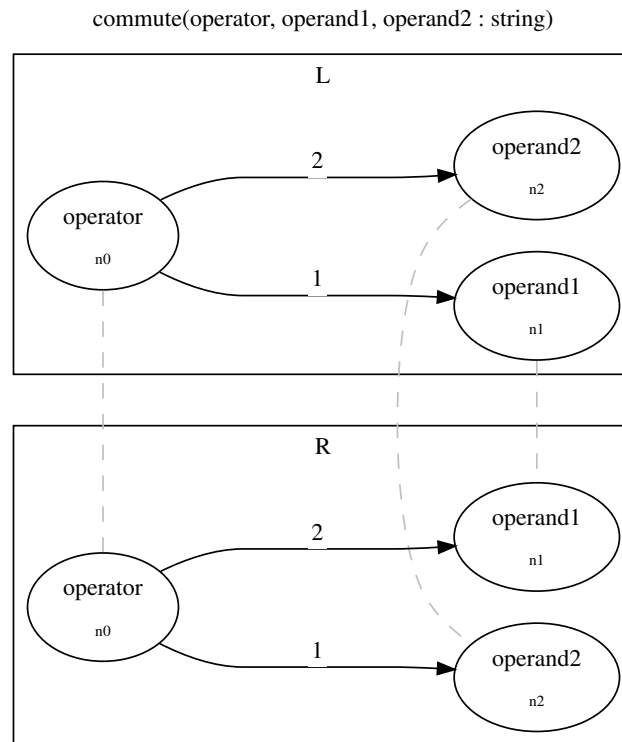
Figure 5 Example GP2 rewrite rule that commutes the operands of a binary operator.

5   Future Work

The next steps in our road map are: commencing the production of instances for selected classes; automatically generate rewrite rules, starting with an initial set of hand-crafted ones; accumulating data on the effects and costs of reformulating the class specifications with a selected collection of rewrite rules; studying the ability of ML models to facilitate solving by selecting good rewrite rules.

Our use of the high-level Essence modelling paradigm enables automated generation of suitable benchmarks for each problem class, using our Generator Instance approach [2, 1]. This system automatically explores the space of valid instances based on the original problem specification expressed in Essence (or in our case, the Emini fragment). The specification is transformed into a parameterised generator instance, and an automatic parameter tuning system is used to identify worthwhile regions of parameter space corresponding to instances of interest.

A key component of our system is the machine learning subsystem that selects rewrite rules. The aim is to select rules to apply to particular classes of problems, on a class level. The lattice of possible rewrite rule sequences is then explored using Monte Carlo tree search. We are currently implementing this aspect of the system.

One approach we are currently exploring is the use of graph embeddings which can isolate and identify particular structures in some vector space, making it amenable to further machine learning operations that work best, or exclusively, on tensor representations. In Figure 6a we show a collection of specifications, automatically produced with a hand-crafted

(a) Abstract Syntax Trees   **(b)** ASTs Embeddings
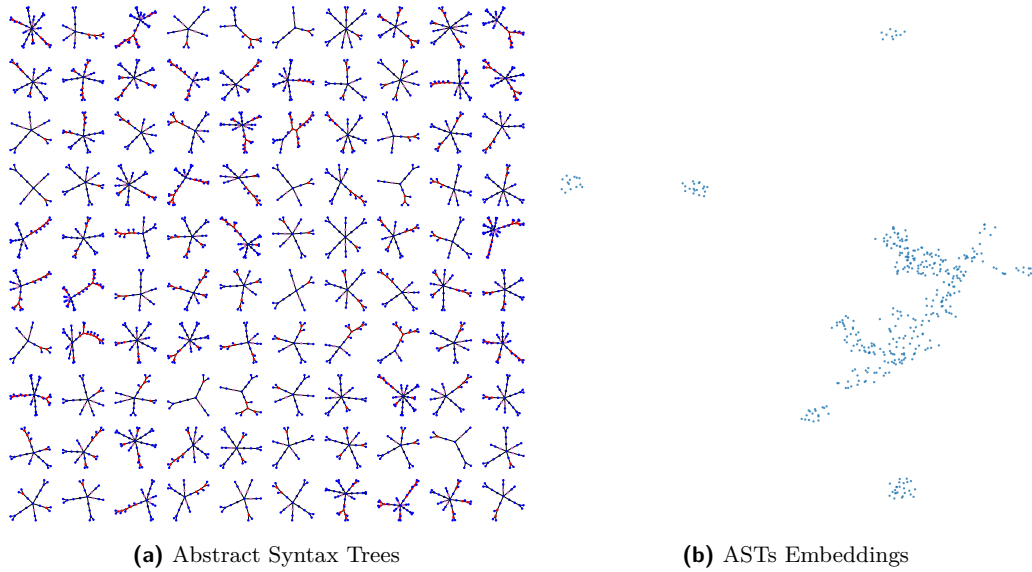
■ **Figure 6** (a): A collection of different specifications in their abstract syntax trees form. Some key elements are highlighted. Red: binary expressions. Navy: decision variables. Pink: letting statements. (b): The ASTs of Figure 6a after embedding. Each dot is a specification, proximity of dots is due to structural similarities.

generator for demonstrative purposes, displaying a variety of different structures. We turn the abstract syntax trees of those specifications into NetworkX graphs which are then embedded into a vector space using an unsupervised technique described in [42]. The results are shown in Figure 6b. These types of embeddings, but even more so supervised ones that take into account the effect on performance of previously tested rewrite rules, can provide metrics that better inform which rewrite rules are likely to benefit a specific class, increasing the efficiency with which these are found.

One of the byproducts of these processes will be the creation of large amounts of semantically equivalent model variants. This information will unlock an important component required to automatically learn and discover new metrics. Building on the idea that the distance between two models can capture their difference, and their proximity captures their sameness, we will provide the data and machinery capable of producing arbitrary amounts of distance zero examples. These components will also benefit those interested in studying the interactions between abstract specifications, reformulations, representations, and solvers.

We expect that improving our tools' ability to recognise that two models that appear different in structure and values are in fact the same or accurately estimating the magnitude of their difference, will enhance their ability to make better choices across the many stages of a problem solving pipeline. These notions will be enriched by the data obtained from solving different reformulations that, even if semantically equivalent, can affect solvers in different ways.

A further virtue of our approach is that rewrite rules able to perform specification strengthening also enable the ability to synthesise altogether new specifications. This provides the ability to autonomously explore new problem classes.

References

1 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, Patrick Spracklen, and Christopher Stone. Discriminating instance generation from abstract specifications: A case study with CP and MIP. *CPAIOR*, LNCS 12296, pages 41–51, 2020. `doi:10.1007/978-3-030-58942-4_3`.

2 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, and Christopher Stone. Instance generation via generator instances. *CP*, LNCS 11802, pages 3–19, 2019. `doi:10.1007/978-3-030-30048-7_1`.

3 Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022. `doi:10.1016/j.artint.2022.103751`.

4 Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z. Salamon. Towards reformulating essence specifications for robustness. *ModRef*, 2021. `doi:10.48550/arXiv.2111.00821`.

5 Robin Arcangioli, Christian Bessiere, and Nadjib Lazaar. Multiple constraint aquisition. *IJCAI*, pages 698–704, 2016. `https://www.ijcai.org/Proceedings/16/Papers/105.pdf`.

6 Saad Attieh, Nguyen Dang, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Athanor: High-level local search over abstract constraint specifications in essence. *IJCAI*, pages 1056–1063, 2019. `doi:10.24963/ijcai.2019/148`.

7 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. *CP*, pages 141–157, 2012. `doi:10.1007/978-3-642-33558-7_13`.

8 Christian Bessiere, Remi Coletta, Abderrazak Daoudi, and Nadjib Lazaar. Boosting constraint acquisition via generalization queries. *ECAI*, pages 99–104, 2014. `doi:10.3233/978-1-61499-419-0-99`.

9 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. *IJCAI*, pages 475–481, 2013. `https://www.ijcai.org/Proceedings/13/Papers/078.pdf`.

10 Christian Bessiere, Remi Coletta, and Thierry Petit. Learning implied global constraints. *IJCAI*, pages 44–49, 2007. `https://www.ijcai.org/Proceedings/07/Papers/005.pdf`.

11 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017. `doi:10.1016/j.artint.2015.08.001`.

12 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 50–53. University of Helsinki, 2020. `https://hdl.handle.net/10138/318754`.

13 Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. `doi:10.1109/TCIAIG.2012.2186810`.

14 Graham Campbell, Jack Romo, and Detlef Plump. The improved GP 2 compiler. *GCM*, pages 206–217, 2020. `https://pure.york.ac.uk/portal/files/71537390/Campbell_Romo_Plump.GCM.20_1.pdf`.

15 John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. *ECAI*, pages 73–77, 2006. `https://ebooks.iospress.nl/volumearticle/2653`.

16 Simon Colton and Ian Miguel. Constraint generation via automated theory formation. *CP*, LNCS 2239, pages 575–579, 2001. `doi:10.1007/3-540-45578-7_42`.

17 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. *AAAI*, pages 7965–7970, 2018. `doi:10.1609/aaai.v32i1.12217`.

18 Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. *LOPSTR*, LNCS 3018, pages 214–232, 2003. `doi:10.1007/978-3-540-25938-1_18`.

19 Eugene C. Freuder. Progress towards the Holy Grail. *Constraints*, 23(2):158–171, 2018. `doi:10.1007/s10601-017-9275-0`.

20 Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. `doi:10.1007/s10601-008-9047-y`.

21 Alan M. Frisch, Christopher Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. The rules of constraint modelling. *IJCAI*, pages 109–116, 2005. `https://www.ijcai.org/Proceedings/05/Papers/1667.pdf`.

22 Alan M. Frisch, Ian Miguel, and Toby Walsh. CGRASS: A system for transforming constraint satisfaction problems. *Recent Advances in Constraints*, LNCS 2627, pages 15–30, 2003. `doi:10.1007/3-540-36607-5_2`.

23 Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. *ECAI*, pages 98–102, 2006. `http://ebooks.iospress.nl/volumearticle/2658`.

24 Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. *FoSSaCS*, LNCS 2030, pages 230–245, 2001. `doi:10.1007/3-540-45315-6_15`.

25 Brahim Hnich. Function variables for constraint programming. *AI Communications*, 16(2):131–132, 2003.

26 Christopher Jefferson and Özgür Akgün. CSPLib: A problem library for constraints. `https://www.csplib.org/`.

27 Zeynep Kiziltan, Marco Lippi, and Paolo Torroni. Constraint detection in natural language problem descriptions. *IJCAI*, pages 744–750, 2016. `https://www.ijcai.org/Proceedings/16/Papers/111.pdf`.

28 Leslie De Koninck, Sebastian Brand, and Peter J Stuckey. Data Independent Type Reduction for Zinc. *Proceedings of the 9th International Workshop on Reformulating Constraint Satisfaction Problems*, 2010.

29 Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia De La Banda. Globalizing constraint models. *CP*, pages 432–447, 2013. `doi:10.1007/978-3-642-40627-0_34`.

30 James Little, Cormac Gebruers, Derek G. Bridge, and Eugene C. Freuder. Using case-based reasoning to write constraint programs. *CP*, page 983, 2003. `doi:10.1007/978-3-540-45193-8_107`.

31 Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008. `doi:10.1007/s10601-008-9041-4`.

32 Patrick Mills, Edward Tsang, Richard Williams, John Ford, and James Borrett. EaCL 1.5: An Easy abstract Constraint optimisation Programming Language. Technical Report CSM-324, Department of Computer Science, University of Essex, 1999.

33 David G. Mitchell and Eugenia Ternovska. Expressive power and abstraction in Essence. *Constraints*, 13(3):343–384, 2008. `doi:10.1007/s10601-008-9050-3`.

34 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. *CP*, LNCS 4741, pages 529–543, 2007. `doi:10.1007/978-3-540-74970-7_38`.

35 Peter Nightingale. Savile Row manual. arXiv, 2021. `doi:10.48550/arXiv.2201.03472`.

36 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. `doi:10.1016/j.artint.2017.07.001`.

37 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. *CP*, pages 590–605, 2014. `doi:10.1007/978-3-319-10428-7_43`.

38 Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. *CP*, pages 330–340, 2015. `doi:10.1007/978-3-319-23219-5_23`.

39   Detlef Plump. From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, 88:154–173, 2017. `doi:10.1016/j.jlamp.2016.12.001`.

40   Reza Rafeh and Negar Jaberi. LinZinc: A library for linearizing Zinc models. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, 40(1):63–73, 2016. `doi:10.1007/s40998-016-0005-1`.

41   Andrea Rendl. *Effective Compilation of Constraint Models*. PhD thesis, University of St Andrews, 2010. `http://hdl.handle.net/10023/973`.

42   Benedek Rozemberczki and Rik Sarkar. Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. *CIKM*, pages 1325–1334, 2020. `doi:10.1145/3340531.3411866`.

43   K. Shchekotykhin and G. Friedrich. Argumentation based constraint acquisition. *ICDM*, pages 476–482, 2009. `doi:10.1109/ICDM.2009.62`.

44   Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023. `doi:10.1016/j.artint.2023.103915`.

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