Automating Reformulation of Essence Specifications via Graph Rewriting

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 ——

Formulating an effective constraint model of a parameterised problem class is crucial to the efficiency with which instances of the class can subsequently be solved. It is difficult to know beforehand which of a set of candidate models will perform best in practice. This paper presents a system that employs graph rewriting to reformulate an input model for improved performance automatically. By situating our work in the ESSENCE abstract constraint specification language, we can use the structure in its high level variable types to trigger rewrites directly. We implement our system via rewrite rules expressed in the Graph Programs 2 language, applied to the abstract syntax tree of an input specification. We show how to automatically translate the solution of the reformulated problem into a solution of the original problem for verification and presentation. We demonstrate the efficacy of our system with a detailed case study.

1 Introduction

Formulating an effective constraint model of a problem of interest is crucial to the efficiency with which the problem can subsequently be solved [6]. It is difficult to know beforehand which of a set of candidate models will perform best in practice. In this paper we present our work in progress on a system that reformulates an input model to improve performance automatically. It differs from some earlier work on automated model reformulation, such as CGRASS [8], Tailor [9], the work of Bessiere et al. [2] to learn implied global constraints, or Savile Row [16], in that it reformulates a model of a parameterised problem class rather than individual problem instances. This has the advantage that the effort made to reformulate the model is amortised over all the instances of the class that are to be solved, rather than having to be paid back during the solution of a single instance. Furthermore, by implementing reformulation as forward-chaining sound rewrite rules, we avoid the need for external verification in existing class-based reformulation [5, 4, 14]. Here we build on our previous proposal for a system [15], demonstrating how to apply graph transformation techniques [21] to achieve automated rewriting of ESSENCE specifications.

In contrast to recent work by Leo et al. [13], who reformulate a low level model of a problem class, we situate our work in the ESSENCE abstract constraint specification language [7]. The advantage of this approach is that the structure apparent in a concise abstract specification can be used to trigger and guide reformulation. For example, Listing 1 presents the k-fold colouring problem we will use in our case study. Here the single abstract decision variable is a binary relation. Having that information directly in the variable type, as

opposed to reconstructing it from a constraint model-level representation likely composed of a constrained matrix of more primitive decision variables, is a significant aid to 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. We implement our system via rewrite rules expressed in the Graph Programs 2 language [3, 18], applied to the abstract syntax tree of an input specification. Given the small size of Essence specifications and the efficiency of the GP2 system in applying graph rewriting rules, the rewriting process has negligible cost. When the type of the decision variable is transformed, the solution must be converted to the original type for verification and presentation to the user. A constraint solver can be employed to solve the solution re-assignment specification straightforwardly. We demonstrate this process even in the case of the nested types supported by Essence.

Our work makes the following primary contributions:

- Automated class-level model reformulation via a library of graph transformation rewrite rules.
- Inverse rewriting of solutions for presentation and verification.
- A case study using k-fold graph colouring.

▮ **Listing 1** Essence specification of k-fold graph colouring, a variant in which each node in a given graph must be assigned k colours with no pair of nodes connected by an edge sharing a colour.

```
$ k-fold graph colouring with k=coloursPerNode, out of numberColours
given n : int
letting vertices be domain int(0..n-1)
given edges : relation (irreflexive) of ( vertices * vertices )
given numberColours : int(1..)
given coloursPerNode : int(1..)
letting colours be domain int(1..numberColours)
find colouring : relation (size n*coloursPerNode) of (vertices * colours)
such that

$ endpoints of edges do not share colours
forAll (u,v) in edges .
    (forAll colourAssignment in colouring .
       (colourAssignment[1] = u) -> !((v,colourAssignment[2]) in colouring)),

$ enforce number of colours per node
forAll u : vertices .
    coloursPerNode = (sum colourAssignment in colouring .
       toInt(colourAssignment[1] = u))
```

2   **Rewriting via graph transformation**

Reformulation can be conceptualised in various ways. In this study, we have chosen to pursue a specific framework for reformulation based on rewriting. The specification at each stage of reformulation can be represented as an abstract syntax tree (AST), from which the specification can be recovered without loss of information. ASTs are commonly modelled as trees, but it is also possible to replace common subtrees in the AST with pointers, transforming the data structure into the form of a directed acyclic graph. In either case, ASTs can be represented as graphs consisting of a set of labelled vertices and a set of directed and labelled arcs between vertices. A node in the AST corresponds to a labelled vertex. A graph rewriting system nondeterministically matches a pattern graph to the target graph; if a match is found, the matched part of the target graph is replaced by a different graph according to the rewrite rule. The details of matching and the types of rewriting may vary, but the graph rewriting paradigm is general enough to be Turing-complete [11] and can thus capture the desired reformulation sequences. Each reformulation in a sequence of reformulations can be considered as a step taken by a graph rewriting system acting on the

AST as the target graph being rewritten. Each reformulation is expressed by a graph rewrite rule or collection of rules. In this study, we have employed the graph rewriting language GP2 [18] to perform rewriting on the abstract syntax tree representing a specification. The GP2 system provides a flexible language for expressing graph rewriting rules and includes an efficient implementation of the rewriting engine optimised for sparse large graphs [3]. The GP2 language can be used to write graph transformation rules, which are compiled using the GP2 engine and become stand-alone C programs that can be executed independently. In the GP2 language, graphs are defined using a set of nodes that are tuples of indices and labels followed by a set of edges that are quadruples of indices, source nodes, target nodes and labels.

3 Reformulating Essence specifications for performance
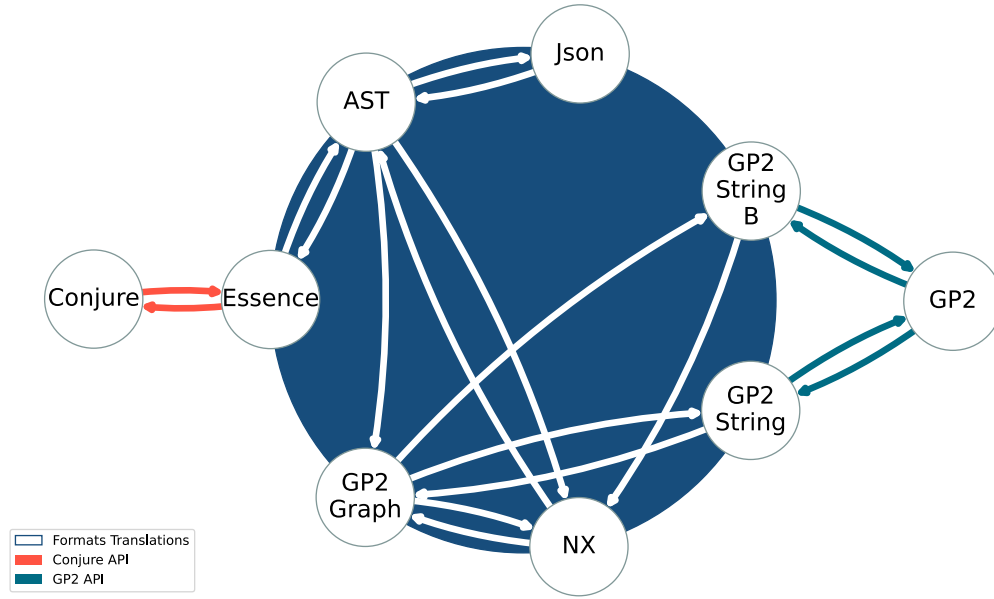

Figure 1 Graph showing format conversions and APIs. The white arrows within the blue area are parts of our system of conversion and the green arrows show interactions with GP2.

The ESSENCE language allows problems to be expressed concisely using high level types such as sets, multisets, functions, relations, and partitions, which may be nested – sets of functions, multisets of relations, etc. ESSENCE specifications are typically solved by a process of *refinement* via the CONJURE tool [1], where the high level types are transformed into collections of more primitive variables, such as matrices of integers, in the solver-independent ESSENCE PRIME language [17]. The SAVILE ROW tool [16] further transforms a model in ESSENCE PRIME into input suitable for a particular constraint, SAT, or SMT solver.

This complex chain of refinement and transformation is sensitive to the ESSENCE specification with which it starts, and so an initial specification may not always lead to optimal solving performance. Therefore, reformulating the specification becomes necessary to improve the efficiency of the solving process. This is often where modellers, through their expertise, can obtain the largest gains in efficiency [20]. One of our objectives is to help automate the application of reformulation rules, sourcing them both from expert modellers or those found via other search methods and building a library of rewriting rules. An advantage

of reformulating ESSENCE is that even a small change can produce a marked change once refinement and parameter substitution has taken place.

The gains provided by reformulations are not only limited to faster solving time but, crucially, can lead to smaller memory footprints across the subsequent refinement steps. In some cases this allows problems to be solved that in their original form may not be able to reach the solver or that could crash during solving.

In practice, a specification written in ESSENCE is turned into GP2 form by a series of conversions and translations by following the shortest path in a conversion graph pictured in Figure 1. We extend our previous work [21], which contains formal definitions for each intermediate form, by adding a different representation of the AST once in GP2 form. In our approach, instead of compounding symbols and their grammatical information into the labels of the GP2 nodes, we add an extra branch to each node separating the two. Once in GP2 form, any of the available precompiled rules can be applied to the AST of the ESSENCE specification. In this particular study, we are interested in a rewrite rule designed to showcase the potential of high-level reformulations making use of relaxation, auxiliary domains, type nesting and constraint rewrites. A fragment of our rule set can be seen in Listing 2, where the left-hand side of the rule is designed to match the portion of the AST in which a relation is declared. The rule takes three parameters, `specName`, `decisionVariableName` and `findPos`, two strings and an integer, respectively, capturing the fact that the labels on those nodes can be any value and can be reused during the rewriting step. Labels in quotes such as `"find"` and `"relation"` must be matched exactly. Nodes in the interface are nodes that will be preserved. In this case, all the nodes are preserved, and only one label is changed by attaching the flag `# red` that will trigger subsequent rules.

◼ **Listing 2** Tagging a decision variable node implemented in GP2.

```
Main = tagRelationDecisionVariable

tagRelationDecisionVariable(specName,decisionVariableName:string;findPos,n:int)
\\ left-hand side of the rule
[
    (n0, specName) (n1, "find") (n2, decisionVariableName) (n3, "relation")
|
    (e1, n0, n1, findPos) (e2, n1, n2, 1) (e3, n2, n3, 1)
]
=>
\\ right-hand side of the rule
[
    (n0, specName) (n1, "find") (n2, decisionVariableName) (n3, "relation"# red)
|
    (e1, n0, n1, findPos) (e2, n1, n2, 1) (e3, n2, n3, 1)
]
interface = {  n0,n1, n2, n3 }
```

4   Case Study: k-fold fractional graph colouring

In order to demonstrate our reformulation system in operation we present a case study with *fractional graph colouring*, which is a generalisation of the graph colouring problem where multiple distinct colours are assigned to each vertex of a graph such that pairs of neighbouring vertices have no colours in common. In the literature it can be found under various different names: Graph Multicoloring Problem, Generalised Graph Colouring Problem, k-Fold Graph Colouring, Fractional Graph Colouring, or Fractional Vertex Colouring. It has been used in the past to showcase constraint programming techniques [10, 19] and it is commonly found in the context of scheduling applications [12]. We will refer to the integer value k in k-fold as the number of colours each node must be assigned, and we will refer to the number of colours available explicitly.

In our case study we take as the input specification that is shown in Listing 1, which employs a binary relation to encode the graph and a binary relation for the decision variable linking each node with the colours assigned to it. This is a natural choice as graphs are commonly modelled using relations, and more than one colour has to be mapped to each node. Two constraints enforce that adjacent vertices do not share the same colour and that the correct number of colours is assigned to each vertex (via a common pseudo-Boolean counting pattern).

From this initial specification, it is possible to activate a sequence of rewriting rules that reformulate the specification with the objective of improving its performance. It can be noted that the number of colours necessary for each vertex is the same for all vertices, the colours assigned to each vertex could be expressed as a set of fixed size, eliminating the need for the counting constraint. This means that one rewrite of the rule should be able to introduce auxiliary domains and another eliminate a constraint. It follows that the decision variable be could turned into a total function that maps each vertex to a set of colours, further removing the need for a quantification, as referring to a set allows referring to all the colours of a specific node in one simpler statement. This requires a rewrite that turns the relation into a function and modifies the existing constraint.

The graph transformation rules proceed as follows:

1. A rewrite rule matches a decision variable of arbitrary name and relation type, that is also mentioned in a quantification with a counting pattern, plus a second relation where the binary relation is over the same object (which is common for graphs) and a quantification over it. Occurrences of the decision variable `colouring` are bold blue in Figure 3.
2. The rule tags the decision type's node, the head of the subtree of the constraint to be eliminated and the inner quantification of the colour that checks that adjacent vertices do not have the same colours. The tagged nodes are highlighted in red in Figure 3.
3. Then, an additional rule propagates the tags in each subtree.
4. Then, a rule is applied until possible to match each pair of tagged nodes; the deeper of the two is removed, and so is the edge between them. This is to avoid deleting a node while leaving a dangling edge.
5. The remaining tagged nodes are used to glue in the desired subtrees, while removing the single node's tags. Three rules match the specific tagged nodes and add the auxiliary domain, the new decision variable and the reformulated constraint.
6. The remaining tagged nodes are removed, together with their parent edge.

■ **Listing 3** Rewritten specification.

```
given n : int(0..100)
letting vertices be domain int(0..n-1)
given edges : relation (irreflexive) of ( vertices * vertices )
given numberColours : int(1..n)
given coloursPerNode : int(1..n)
letting colours be domain int(1..numberColours)

letting coloursSet be domain set (size coloursPerNode) of colours

find colouring : function (total) vertices --> coloursSet
such that
forAll (u,v) in edges .
    colouring(v) intersect colouring(u) = {}
```

This procedure adds an auxiliary domain of type `set` that will host the colours, a new decision variable of type `function` mapping the left-hand side of the relation to the auxiliary set, and shrinks the constraints to simply require an empty intersection of two sets. The final product can be seen in Listing 3.

```
given n : int
letting vertices be domain int(0..n-1)
given edges : relation (irreflexive) of ( vertices * vertices )
given numberColours : int(1..)
given coloursPerNode : int(1..)
letting colours be domain int(1..numberColours)
find colouring : relation (size n*coloursPerNode) of (vertices * colours)
such that
forAll (u,v) in edges .
    (forAll colourAssignment in colouring .
        (colourAssignment[1] = u) -> !((v,colourAssignment[2]) in colouring))
,
forAll u : vertices .
    coloursPerNode = (sum colourAssignment in colouring .
        toInt(colourAssignment[1] = u))
```
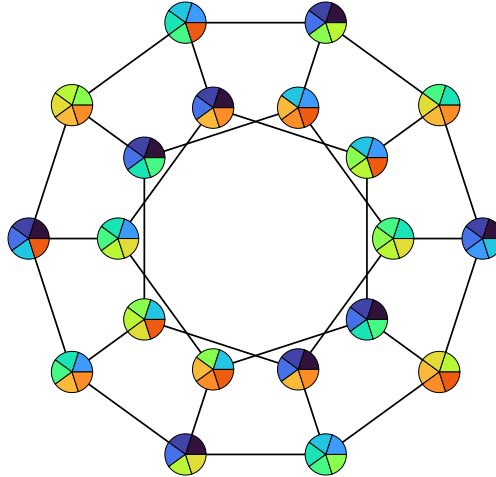
■ **Figure 2** Parts of the specification that are interacted with during rewrite. Matched subtrees are highlighted in light blue. Names that can be used as free parameters are coloured in dark blue. Tagged nodes are highlighted in red.

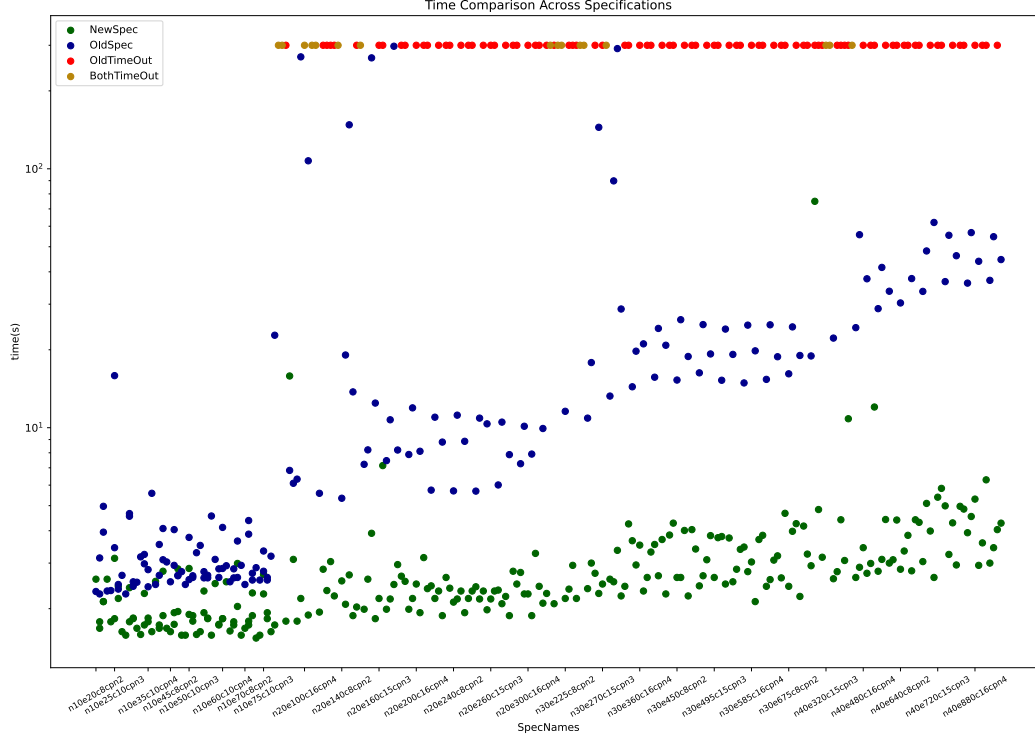## 4.1   Empirical evaluation of the reformulation

In order to test the performance of the reformulated abstract specification we have developed an instance generator that produces input parameters; this uses a simple grid search spanning a range of values for n nodes and e edges for the graph to be coloured, c colours available and cpn colours per nodes enforced. Values. n: 10-40, step size 10. e: $n^2 * 20\% - 80\%$, step size 5%. cpn: 2-5, step size 1. c: cpn*4-6, step size 1.

Each instance is solved using both models and the results are plotted in Figure 4.

Our system can also translate graphs from the Python library NetworkX into Essence relations, which allows the conversion of any of its library of graphs into instances by supplying the c and cpn parameters. A small selection of iconic graphs has been used for additional testing, such as obtaining the 5-fold colouring of the dodecahedral graph in Figure 3.

■ **Figure 3** 5-fold colouring of the dodecahedral graph.

Figure 4 Performance of the two specifications: we plot instances (with selected names listed along the horizontal axis), the reformulated specifications are green, the old specifications are blue, gold indicates both specifications time out, red indicates that the old specification times out.
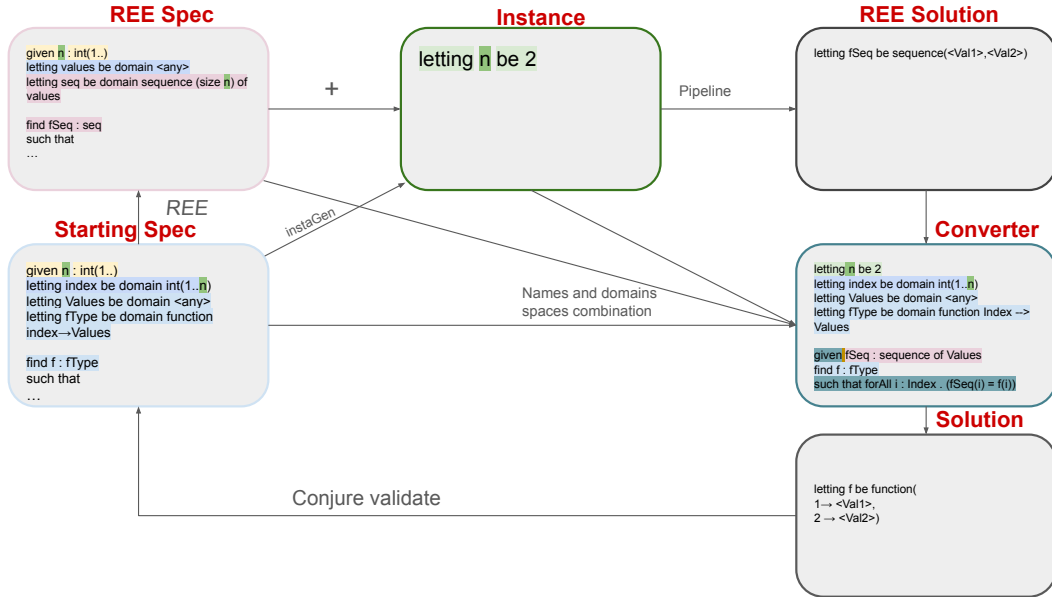
5 Converting solutions to their original type

When a reformulation changes the type of a decision variable, the solution produced by the solver will be of the transformed type. To verify that the solution of the reformulated specification is, in fact, a solution to the original problem, we must convert it back to its original type. To achieve this, we create a converter based on an ESSENCE specification, which requires the solution of the reformulated problem as a parameter. The converter (illustrated in Listing 4) includes all the variables declared in the original specification, along with all of its parameters. Additionally, it must have a decision variable with exactly the same name as the original specification, and a constraint mapping assignments of the new type to the old type. Solving this parameterised specification produces a solution that follows the structure of the original type that can be checked using CONJURE with the `-validate-solution` option.

We timed the conversion of all the solutions obtained by solving the instances produced in the previous section. The worst-performing conversion took 0.56 seconds, with an overall average of 0.459 seconds. This overhead is significant only for the smallest instances.

Listing 4 Solution Converter

```
given n : int(0..100)
given numberColours, coloursPerNode : int(1..n)
letting vertices be domain int(0..n-1)
letting colours be domain int(1..numberColours)
letting coloursSet be domain set (size coloursPerNode) of colours
given solution : function (total) vertices --> coloursSet
find colouring : relation (size n*coloursPerNode) of ( vertices * colours )
such that forAll item in defined(solution) . forAll colour in solution(item) . colouring(item, colour)
```

Figure 5 Specification of the solution's converter and production flow. In this simpler example a function indexed by integers is turned into a sequence and back again.

6   Conclusion

In this paper we have demonstrated an automated system for the reformulation of ESSENCE specifications. Our system, implemented in the Graph Programs 2 language, reformulates specifications of parameterised problem classes, giving the key advantage of reformulation effort being amortised over the set of instances from the class to be solved. High-level ESSENCE specifications are concise and highly structured, aiding rewriting. Furthermore, they are not tied to any particular solving paradigm, hence specification reformulations may be tested against a variety of modelling and solving choices.

Future work. Thanks to this foundational work, we have created the basis necessary to develop a system where the rewrite rules are produced automatically, or semi-automatically, starting from pairs of specifications that express possible reformulations. Automating the creation of rewrite rules would not only free the developer from the need to write GP2 programs but also collect rewrites from past specifications that have been refined over time in order to populate a rewriting library by distilling previous expert knowledge sitting idle in existing catalogues and literature. Another enticing future direction is the production of rewrite rules using search methods, which could be applied to any arbitrary specification. The aim would be to produce reformulations that improve the current model under consideration and are reusable.

References

1   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.

2   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.

3 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`.

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

5 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`.

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

7 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`.

8 Alan M Frisch, Ian Miguel, and Toby Walsh. CGRASS: a system for transforming constraint satisfaction problems. *International ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 15–30. Springer, 2002. `doi:10.1007/3-540-36607-5_2`.

9 Ian P Gent, Ian Miguel, and Andrea Rendl. Tailoring solver-independent constraint models: A case study with Essence' and Minion. *Abstraction, Reformulation, and Approximation: 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21*, pages 184–199. Springer, 2007. `doi:10.1007/978-3-540-73580-9_16`.

10 Stefano Gualandi and Federico Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, 2012.

11 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`.

12 Klaus Jansen and Lorant Porkolab. Preemptive scheduling with dedicated processors: applications of fractional graph coloring. *Journal of Scheduling*, 7:35–48, 2004.

13 Kevin Leo, Grame Gange, Maria Garcia de la Banda, and Mark Wallace. Automatic core-guided reformulation via constraint explanation and condition learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8065–8072, Mar. 2024. `doi:10.1609/aaai.v38i8.28645`.

14 Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Globalizing constraint models. *Artificial Intelligence*, 302:103599, 2022.

15 Ian Miguel, András Salamon, and Christopher Stone. Towards exploratory reformulation of constraint models. *PTHG-23: The Sixth Workshop on Progress Towards the Holy Grail*, 2023. `https://research-repository.st-andrews.ac.uk/handle/10023/28735`.

16 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`.

17 Peter Nightingale and Andrea Rendl. Essence' description, 2016. `arXiv:1601.02865`.

18 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`.

19 Steven Prestwich. Generalised graph colouring by a hybrid of local search and constraint programming. *Discrete Applied Mathematics*, 156(2):148–158, 2008.

20 Jean-Charles Régin. Simple solutions for complex problems: ACP Research Excellence Award talk, 2013. Presented at CP.

21 Christopher Stone, András Z. Salamon, and Ian Miguel. A graph transformation-based engine for the automated exploration of constraint models. *International Conference on Graph Transformation*, volume 14774 of *LNCS*. Springer, 2024.