Automatically Generating Streamlined Constraint Models with Essence and Conjure

James Wetter, Özgür Akgün, and Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, UK
{jpw3, ozgur.akgun, ijm}@st-andrews.ac.uk

Abstract. Streamlined constraint reasoning is the addition of uninferred constraints to a constraint model to reduce the search space, while retaining at least one solution. Previously, effective streamlined models have been constructed by hand, requiring an expert to examine closely solutions to small instances of a problem class and identify regularities. We present a system that automatically generates many conjectured regularities for a given Essence specification of a problem class by examining the domains of decision variables present in the problem specification. These conjectures are evaluated independently and in conjunction with one another on a set of instances from the specified class via an automated modelling tool-chain comprising of Conjure, Savile Row and Minion. Once the system has identified effective conjectures they are used to generate streamlined models that allow instances of much larger scale to be solved. Our results demonstrate good models can be identified for problems in combinatorial design, Ramsey theory, graph theory and group theory - often resulting in order of magnitude speed-ups.

1 Introduction

The search space defined by a constraint satisfaction problem can be vast, which can lead to impractically long search times as the size of the problem instance considered grows. An approach to mitigating this problem is to narrow the focus of the search onto promising areas of the search space using streamliners [11]. Streamliners take the form of uninferred additional constraints (i.e. constraints not proven to follow from the original problem statement) that rule out a substantial portion of the search space. If a solution lies in the remainder then it can typically be found more easily than when searching the full space. Previously, streamlined models have been produced by hand [11,13,15,16], which is both difficult and time-consuming. The principal contribution of this paper is to show how a powerful range of streamliners can be generated automatically.

Our approach is situated in the automated constraint modelling system Conjure [2]. This system takes as input a specification in the abstract constraint specification language Essence [7,8]. Figure 1 presents an example specification, which asks us to partition the integers $1 \dots n$ into k parts subject to a set of constraints. Essence supports a powerful set of type constructors, such as set, multi set, function and relation, hence Essence specifications are concise and

```
language Essence 1.3

given k, l, n : int(1..)

find p: partition (numParts k) from int(1..n)
such that
 forAll s in parts(p) .
  forAll start : int(1..n-l+1) .
   forAll width : int(1..(n-start+1)/(l-1)) .
    !(forAll i : int(0..l-1) .
       (start + i*width) in s)
```

Fig. 1: Essence specification of the Van Der Waerden Number Problem [24] (see Appendix A). The specification describes a certificate for the lower bound on $W(k, l)$, and will be unsatisfiable if `n` is equal to $W(k, l)$.

highly structured. Existing constraint solvers do not support these abstract decision variables directly. Therefore we use Conjure to *refine* abstract constraint specifications into concrete constraint models, using constrained collections of primitive variables (e.g. integer, Boolean) to represent the abstract structure.

Our method exploits the structure in an Essence specification to produce streamlined models automatically, for example by imposing streamlining constraints on or between the parts of the partition in the specification in Figure 1. The modified specification is refined automatically into a streamlined constraint model by Conjure. Identifying and adding the streamlining constraints at this level of abstraction is considerably easier than working directly with the constraint model, which would involve first recognising (for example) that a certain collection of primitive variables and constraints together represent a partition — a potentially very costly step.

Our system contains a set of rules that fire when their preconditions match a given Essence specification to produce candidate streamliners. Using Conjure to refine the streamlined specifications into constraint models, solved with Savile Row[1] [19] and Minion[2] [10], candidates are evaluated against instances of the specified class. Effective streamliners are combined to produce more powerful candidates. As we will show, high quality streamlined models can be produced in this way, in some cases resulting in a substantial reduction in search effort.

Herein we focus on satisfaction problems. Optimisation is an important future consideration but requires different treatment: streamliners may allow us to find good solutions quickly but exclude the optimal solution to the original model.

The rest of this paper is structured as follows. Following a summary of related work, we give some background on the Essence language. Section 4 describes in detail our approach to generating streamliners automatically, then Section

[1] `http://savilerow.cs.st-andrews.ac.uk`
[2] `http://constraintmodelling.org/minion/`

5 discusses combining streamliners to produce yet more effective models. We conclude following a discussion of discovering streamliners in practice.

2 Related Work

Colton and Miguel [5] and Charnley et al [4] used Colton's HR system [6] to conjecture the presence of implied constraints from the solutions to small instances of several problem classes, including quasigroups and moufang loops. The Otter theorem prover[3] was used to prove the soundness of these conjectures. If proven sound, the implied constraints were added to the model to aid in the solution of larger instances of the same problem class.

Streamlined constraint reasoning differs from the approach of Charnley et al in that the conjectured constraints are not typically proven to follow from a given constraint model. Rather, they are added in the hope that they narrow the focus of the search onto an area containing solutions. When first introduced by Gomes and Sellmann [11] streamlined constraint reasoning was used to help construct diagonally ordered magic squares and spatially balanced experiment designs. For the magic squares the additional structure enforced was regularity in the distribution of numbers in the square. That is, the small numbers are not all similarly located, and likewise for large numbers. The spatially balanced experiment designs were forced to be self-symmetric: the permutations represented by the rows of the square must commute with each other. For both of these problems streamlining led to huge improvements in solve time, allowing much larger instances to be solved.

Kouril et al. refer to streamlining as "tunneling" [13]. They describe the additional constraints as tunnels that allow a SAT solver to tunnel under the great width seen early in the search tree when computing bounds on Van de Waerden numbers. They used simple constraints that force or disallow certain sequences of values to occur in the solutions. Again this led to a dramatic improvement in run time of the solver, allowing much tighter bounds to be computed.

Le Bras et al. used streamlining to help construct graceful double wheel graphs [15]. Constraints forcing certain parts of the colouring to form arithmetic sequences allowed for the construction of colourings for much larger graphs. These constraints led to the discovery of a polynomial time construction for such colourings, proving that all double wheel graphs are graceful.

Finally Le Bras et. al. made use of streamlining constraints to compute new bounds on the Erdős discrepancy problem [16]. Here constraints enforcing periodicity in the solution, the occurrence of the improved Walters sequence, and a partially multiplicative property improved solver performance, allowing the discovery of new bounds.

In all of these examples streamliners proved very valuable, but were generated by hand following significant effort by human experts. In what follows, we will show that the structure recognised and exploited by these experts is often present in abstract constraint specifications.

[3] `http://www.cs.unm.edu/~mccune/otter/`

```
language Essence 1.3

given v, b, r, k, lambda : int(1..)
where v = b, r = k
letting Obj   be new type of size v,
        Block be new type of size b

find bibd : relation (symmetric) of (Obj * Block)

such that
  forAll o  : Obj   . |bibd(o,_ )| = r,
  forAll bl : Block . |bibd(_,bl)| = k,
  forAll o1, o2 : Obj .
    o1 != o2 -> |bibd(o1,_) intersect bibd(o2,_)| = lambda
```

Fig. 2: Essence specification of the square (ensured by the `where` statement) Balanced Incomplete Block Design Problem [20]. Streamliner added as an Essence annotation shown underlined.

3 Background: Essence

The motivation for abstract constraint specification languages, such as Zinc [18] and Essence is to address the *modelling bottleneck*: the difficulty of formulating a problem of interest as a constraint model suitable for input to a constraint solver. An abstract constraint specification describes a problem above the level at which constraint modelling decisions are made. An automated refinement system, such as Conjure, can then be used to produce a constraint model from the specification automatically.

An Essence specification, such as those given in Figures 1 and 2, identifies: the input parameters of the problem class (`given`), whose values define a problem instance; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy (`such that`). In addition, an objective function may be specified (`min/maximising` — not shown in these examples) and identifiers declared (`letting`). Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. Our Conjure system employs refinement rules to convert an Essence specification into the solver-independent constraint modelling language Essence′. From Essence′ we use Savile Row to translate the model into input for a particular solver while performing solver-specific model optimisations.

A key feature of abstract constraint specification languages is the 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. This allows the problem structure to be captured very concisely. As explained below, this clarity of structure is a good basis for the conjecture of streamlining constraints.

4 From Conjectures to Streamlined Specifications

This section presents the methods used to generate streamlined models automatically. The process is driven by the decision variables in an Essence specification, such as the partition in Figure 1. For each variable, the system forms conjectures of possible regularities that impose additional restrictions on the values of that variable's domain. Since the domains of Essence decision variables have complex, nested types, these restrictions can have far-reaching consequences for constraint models refined from the modified specification. The intention is that the search space is reduced considerably, while retaining at least one solution. Multiple conjectures found to produce successful streamlined specifications individually can be combined in an attempt to produce a single more sophisticated streamliner. Currently conjectures are formed about each variable independently; an important future direction is to make conjectures across multiple variables.

4.1 Exploiting Essence Domain Annotations

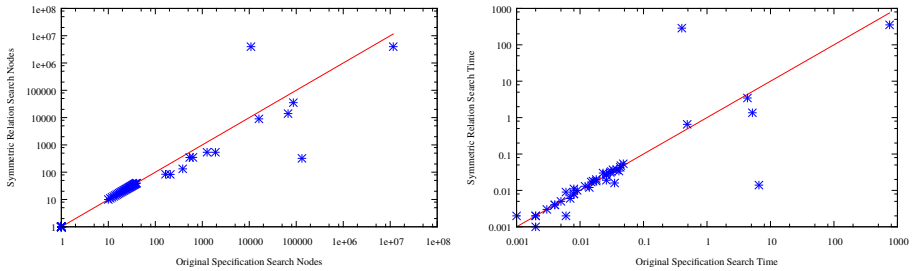
Essence allows domains to be annotated to restrict the set of acceptable values. For example, a function variable domain may be restricted to injective functions, or a partition variable domain may be restricted to regular partitions. Hence, the simplest source of streamliners is the systematic annotation of the decision variables in an input specification. This sometimes retains solutions to the original problem while improving solver performance. Consider the Balanced Incomplete Block Design problem (BIBD, Figure 2), where the decision variable is a relation. For square BIBDs we might consider a streamliner requiring a symmetric relation, achieved simply by adding the `symmetric` annotation as shown.

Figure 3 summarises an experiment with this streamliner on a set of satisfiable square BIBD instances. Original and streamlined specifications were refined with Conjure, using the Compact heuristic [1] to select one model. For each instance Savile Row was used to prepare the resulting model for Minion, which was used to find a solution. Streamlining uniformly resulted in a solvable instance, and as seen in Figure 3a in all but one instance search size is equivalent or reduced and as seen in Figure 3b the corresponding execution time is sometimes reduced.

4.2 Conjecture-forming Rules

The existing Essence domain annotations are, however, of limited value. They are very strong restrictions and so often remove all solutions to the original problem when added to a specification. In order to generate a larger variety of useful conjectures we employ a small set of rules, categorised into two classes:

1. First-order rules add constraints to reduce the domain of a decision variable directly.
2. Higher-order rules take other rules as arguments and use them to reduce the domain of a decision variable.



(a) Search nodes. (b) Search time.

Fig. 3: Search effort to find the first solution to satisfiable square BIBD instances where $\mathtt{v} = \mathtt{b} \leq 40$: original vs streamlined specification.

The full list of rules is shown in Table 1. A selection of the first-order rules is given in Figure 4, and a selection of the higher-order rules are given in Figure 5.

We define four first-order rules that operate on integer domains directly: 'odd', 'even', 'lowerHalf' and 'upperHalf'. Each restricts an integer domain to an appropriate subset of its values. We define six first-order rules for function domains. Two of these constrain the function to be monotonically increasing (or decreasing). The other four place the largest (or smallest) element first (or last). Functions mapping a pair of objects to a third object of the same type

Class	Trigger Domain	Name	Softness
First-order	`int`	odd{even}	no
		lower{upper}Half	no
	`function int -> int`	monotonicIncreasing{Decreasing}	no
		largest{smallest}First{Last}	no
	`function (X,X) -> X`	commutative	no
		associative	no
		non-commutative	no
	`partition from X`	quasi-regular	yes
Higher-order	`set of X`	all	no
		most	yes
		half	no
		approxHalf	yes
	`function X -> Y`	range	no
		defined	no
		pre{post}fix	yes
		allBut	yes
	`function (X,X) -> Y`	diagonal	no
	`partition from X`	parts	no

Table 1: The rules used to generate conjectures. Rows with a softness parameter specify a family of rules each member of which is defined by an integer parameter.

Name odd
Input `X: int`
Output `X % 2 = 1`

Name lowerHalf
Input `X: int(l..u)`
Output `X < l + (u - l) / 2`

Name monotonicIncreasing
Input `X: function int --> int`
Output
```
forAll i in defined(X) .
 forAll j in defined(X) .
   i < j -> X(i) <= X(j)
```

Name largestFirst
Input `X: function int(l..u) --> int`
Output
```
forAll i in defined(X) .
 X(min(defined(X)) <= X(i)
```

Name commutative
Input `X: function (D, D) --> D`
Output
```
forAll (i,j) in defined(X) .
 X((i,j)) = X((j,i))
```

Name quasi-regular
Input `X: partition from _`
Output
```
minPartSize(X,
 |participants(X)|/|parts(X)| - k)
   /\
maxPartSize(X,
 |participants(X)|/|parts(X)| + k)
```

Fig. 4: A selection of the first-order streamlining rules.

Name all
Parameter `R` (another rule)
Input `X: set of _`
Output `forAll i in X . R(i)`

Name most
Parameter `R` (another rule)
Parameter `k` (softness)
Input `X: set of _`
Output
```
k >= sum i in X . toInt(!R(i))
```

Name range
Parameter `R` (another rule)
Input `X: function _ --> _`
Output `R(range(X))`

Name prefix
Parameter `R` (another rule)
Parameter `k` (softness)
Input `X: function int(l..h) --> _`
Output `R(restrict(X, ‘int(l..h-k)‘))`

Name parts
Parameter `R` (another rule)
Input `X: partition of _`
Output `R(parts(X))`

Name diagonal
Parameter `R` (another rule)
Input `X: function (D1, D1) --> D2`
Output
```
{ R(X')
@ find X' : function D1 --> D2
  such that
   forAll i : D1 .
    (i,i) in defined(X) -> X'(i) = X((i,i)),
   forAll i : D1 .
    i in defined(X') -> X'(i) = X((i,i))
}
```

Fig. 5: A selection of the higher-order streamlining rules.

can be viewed as binary operators on these objects. We define three first-order rules to enforce such functions to be commutative, non-commutative, and associative respectively. Finally, we define a first-order rule for partitions called 'quasi-regular'. Partition domains in Essence can have a `regular` annotation, however this can be too strong. The 'quasi-regular' streamlining rule posts a soft version of the regularity constraint, which takes an integer parameter, `k`, to control the softness of the constraint. In our experiments we varied the value of `k` between 1 and 3. Larger values of `k` will make the constraint softer as they allow the sizes of parts in the partition to be `k`-apart.

Higher-order rules take another rule as an argument and lift its operation to a decision variable with a nested domain. For example, the 'all' rule for sets applies a given streamlining rule to all members of a set, if it is applicable. We define three other higher-order rules that operate on set variables: 'half', 'most' and 'approxHalf', the last two with softness parameters. For a set of integers, applying the 'half' rule with the 'even' rule as the parameter – denoted 'half(even)' – forces half of the values in the set to be even. The parameter to the higher-order rule can itself be a higher order rule, so for a set of set of integers 'all(half(even))' constrains half of the members of all inner sets to be even.

The 'defined' and 'range' rules for functions use the defined and range operators of Essence to extract the corresponding sets from the function variable. Once the set is extracted the parameter rule `R` can be directly applied to it. The 'prefix' and 'postfix' rules work on functions that map from integers by focusing on a prefix or postfix of the integer domain. The 'parts' rule views a partition as a set of sets and opens up the possibility of applying set rules to a partition.

The 'diagonal' rule introduces an auxiliary function variable. The auxiliary variable represents the diagonal of the original variable, and it is channelled into the original variable. Once this variable is constructed the streamlining rule taken as a parameter, `R`, can be applied directly to the auxiliary variable. Similarly the 'allBut' rule introduces an auxiliary function variable that represents the original variable restricted to an arbitrary subset (of fixed size) of its domain. This is similar to the 'prefix' rule but allows the ignored mappings of the function to fall anywhere in the function's domain rather than only at the end.

It is important to note that allowing higher-order rules to take other higher-order rules as parameters naively can lead to infinite recursion; such as 'prefix(prefix(prefix(...)))' or 'prefix(postfix(prefix(..)))'. We considered two ways of mitigating this problem: 1) using a hard-coded recursion limit 2) only allowing one application of the same higher-order rule in a chain of rule applications and at the same level. Using a hard-coded recursion limit has two drawbacks. It still allows long lists of useless rule applications like applying 'prefix' repeatedly. It can also disallow useful but long sequences of rule applications. Instead, we implemented a mechanism where the same higher-order rule can only be applied once at the same level; that is, a rule can only be applied more than once if the domain of its input is a subcomponent of the domain of the input of the previous application. This disallows 'prefix(prefix(...))', but allows 'pre-

fix(range(half(prefix(even),1)),1)' which works on a decision with the following type `function int -> (function int -> int)`.

In order to apply these rules to Essence problem specifications, we extract a list of domains from the top level decision variables. For each domain, we find the list of all applicable rules. Each application of a streamlining rule results in a conjecture that can be used to streamline the original specification.

To see an example of an effective constraint generated by this system consider the problem of generating certificates for lower bounds on Van Der Waerden numbers shown in Figure 1. Here only one decision variable is present, a partition of integers, `p`. First the rule *parts* is triggered so `p` is treated as a set of set of integers. Next, the rule *all* is triggered such that all the members of the outer set are restricted, then another rule *approxHalf* is triggered so approximately half the members of each inner set are restricted. Finally *lowerHalf* is triggered so the domain of the integer has its upper bound halved. The complete rule that is being applied is 'parts(all(approxHalf(lowerHalf, i)))', and the resulting constraint is:

```
forAll s in parts(p) .
  |s|/2 + i >= sum x in s . toInt(x <= n/2) /\
  |s|/2 - i <= sum x in s . toInt(x <= n/2)
```

where `i` is the parameter given to *approxHalf*. This constraint enforces that each part of the partition consists of approximately half 'small' numbers and half 'large' numbers. Figure 8 shows that this constraint with $\texttt{i} = 2$ drastically reduces the number of search nodes explored when finding a single solution to the problem.

Initially these rules were applied to the variable domains declared as finds in the original specification. It was observed that the wheel like structures in the graceful graph labeling problems were not exposed in the function variables used to represent the labeling. In order to extract such structures a preprocessing step is performed that introduces restricted function variables to the rule system in order to generate a wider range of streamlining constraints.

This preprocessing step is triggered when a function variable, $f : A \to B$, is present in the find declarations. For each such variable the constraints are checked for an application of the function, $f(a)$, quantified over a strict subset of the domain, $a \in A' \subset A$. The quantification can be $\sum$, $\forall$ or $\exists$. If such an expression is present in the constraints the conjecture generation rules are also applied to a restricted version of the function, $f|_{A'}$.

This system is capable of generating novel streamliners that have not been previously reported, such as the 'quasi-regular' rule. In addition it generates some streamliners very similar to some of those previously seen in the literature. For example appliying 'restrict(range(all(odd)),C_1)' to the colouring function in the gracful double wheel labelling specification defines the same solution set as 'C_1 is odd' in La Bras et. al. [16]. Although it should be noted that the rules presented here do not generate all streamliners previously reported.

Problem Class		Total	Retain	Improve
Graceful Graph Colouring [21]	Wheel Graphs [9]	1479	1299	136
	Double Wheel Graphs [15]	2142	1942	466
	Helm Graphs [3]	1428	1296	85
	Gear Graphs [17]	1428	1214	70
Quasigroup Existence [23]	QGE3	593	570	51
	QGE4	593	572	34
	QGE5	593	582	24
	QGE6	593	569	44
	QGE7	593	555	32
Equidistant Frequency Permutation Arrays [12]		560	377	2
Van Der Waerden Numbers [24]		433	364	14
Schur Numbers [22]		437	419	0
Langfords Problem [14]		357	228	24

Table 2: Number of conjectures generated for a set of problem classes. The `Total` column lists all conjectures generated for each class, the `Retain` column lists the number of conjectures that retain at least one solution and the `Improve` column lists the number of conjectures that improve solver performance.

4.3 Experimental Analysis

To evaluate this system it was used to streamline several different problem classes, consisting of graceful graph colouring problems [21], quasigroup existence problems [23], equidistant frequency permutation arrays [12], Van Der Waerden numbers [24] with $k = 3$ and $l = 4$, Schur numbers [22] and Langford's problem [14] (see Appendix A). The instances used to evaluate the performance of the streamlined models were obtained by increasing the size of the integer parameters until the original specification was unable to compute one solution in under 100,000 search nodes. Experiments were run on a 32-core AMD Opteron 6272 at 2.1 GHz and took around 5 hours per problem class to complete.

Table 2 shows the total number of conjectures generated, the number of conjectures that retain solutions to the original problem and the number of conjectures that reduce the total number of search nodes explored after the specification is refined, tailored and solved for all instances.

Figure 6 shows the search nodes and solve time to find the first solution for the original specification vs the best streamlined model, where the quality metric compares the number of instances solved in under 100,000 search nodes and breaks ties by comparing the total number of search nodes across all instances. Both the size of the search and search time are often reduced by orders of magnitude by the streamlining constraints.

5 Identifying Effective Combinations of Conjectures

It has previously been observed that applying several streamlining constraints to a model simultaneously can result in larger performance gains that any of the

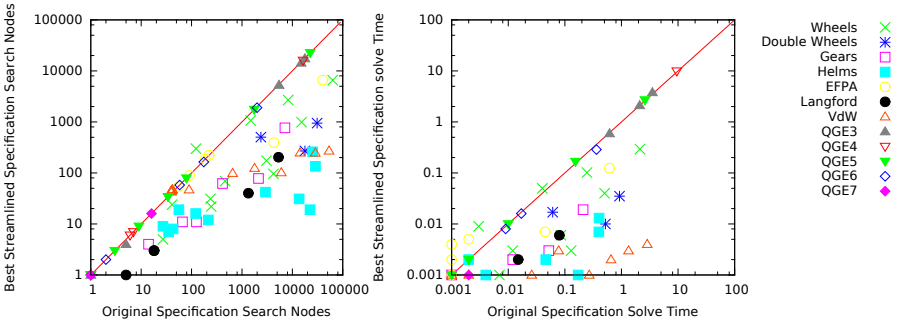


Fig. 6: The size and execution time of search required to find the first solution for a collection of problem classes for both the original specification and the streamlined specification that resulted in the smallest cumulative search size across instances. The generated conjectures often result in order of magnitude reduction in search size for harder problem instances.

constraints in isolation [15]. In order to find such combinations of constraints we must consider the power set of constraints that retain solutions to the original problem. In this section we investigate finding powerful combinations of constraints automatically with the use of pruning and make a comparison between depth first search and breadth first search of the lattice of constraints.

For many of the problems considered here a large number of singleton conjectures that retain solutions are generated (see Table 2) resulting in power sets too large to be exhaustively explored in practice. Two forms of pruning were used to reduce the number of combinations to be considered:

1. if a set of conjectures fails all supersets are excluded from consideration (see Figure 7),
2. trivially conflicting conjectures are not combined, for example we avoid forcing a set to simultaneously contain only odd numbers and contain only even numbers. We associate a set of tags with each of the rules in order to implement this pruning. Rules applied to the same variable that share tags are not combined. This also removes the possibility of combining two different conjectures that differ only by a softness parameter.

These pruning rules only remove combinations that are sure to fail, or are equivalent to a smaller set of conjectures.

Even with this pruning the number of combinations to consider was found to be too large to allow exhaustive enumeration. Therefore a traversal of the lattice allowing good combinations to be identified rapidly is desired. Here we experimented with depth first search (DFS) and breadth first search (BFS) of the lattice. In order to guide both the searches the singleton conjectures were ordered from best to worst, where the quality metric compared the number of instances

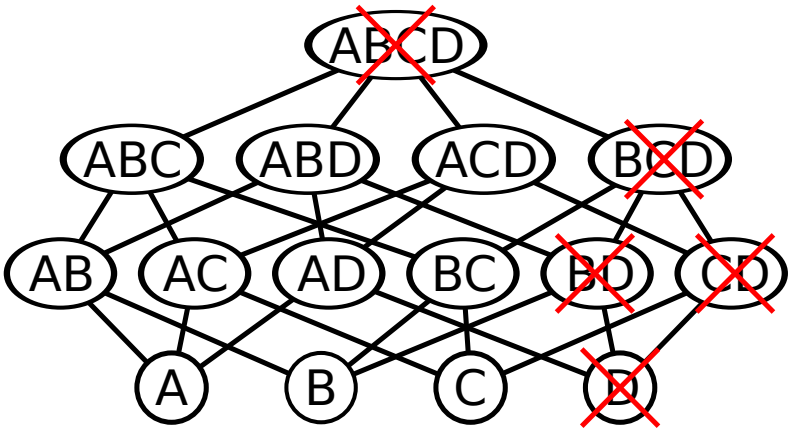


Fig. 7: The power set of singleton conjectures can be explored to identify combinations that result in powerful streamlined specifications. If small sets of conjectures that fail to retain solutions are identified all super sets can be pruned from the search vastly reducing the number of vertices to be explored.

solved in under 100,000 search nodes and ties were broken by comparing the total search nodes across all instances.

In order to compare the two approaches they were each allowed to evaluate 500 combinations of conjectures in addition to the singleton conjectures already evaluated for three problem classes (Van der Waerden numbers, graceful helms graphs and graceful double wheel graphs). The set of instances used for evaluation was augmented by increasing the integer parameters until the best singleton streamliner was unable to solve the instance in under 100,000 search nodes. Each combination of conjectures was evaluated by refining, tailoring and solving for the first solution using Conjure's compact heuristic and default settings for Savile Row and Minion. The experiments were performed on 32-core AMD Opteron 6272 at 2.1 GHz taking approximately 6 hours for each problem class.

Figures 8 to 10 show the best set of conjectures found by this process, where the quality metric compares the number of instances solved and ties were broken by comparing the total number of search nodes.

Figure 8 shows three singleton conjectures that were found to produce an effective streamlined specification of the Van Der Waerden numbers problem, one of which results from the chained application of four rules, whereas another results directly from a single rule. Figure 9 shows two conjectures being combined for the graceful helm graph problem, one of which results from the chained application of three rules. Figure 10 shows the combination of two conjectures for the graceful colouring of double wheel graphs. In all cases the combination of streamliners results in better performance than any of the streamliners in isolation.

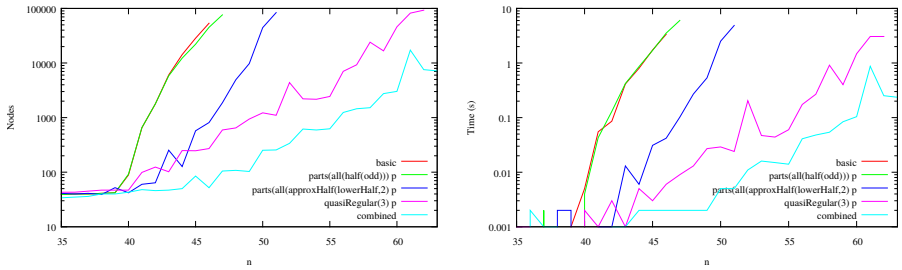


Fig. 8: Combining singleton conjectures to produce a more effective streamlined model for Van der Waerden numbers. The instances have `k = 3`, `l = 4` and `n` varies. The first conjecture ensures odd number are evenly distributed between the parts of the partition. The second conjecture ensures the 'small' numbers are evenly distributed between the parts of the partition. The third conjecture ensures the sizes of the partition vary from each other by at most three.

On one problem, Van Der Waerden numbers, DFS performed very well, reaching a powerful set of three conjectures within the first three models evaluated. On the other problems DFS performed poorly, unable to beat the best pair found by BFS within the allotted resource budget. In all three cases DFS failed to find an improved model after the first five models it considered.

The poor performance of DFS can be attributed to two factors. First, the two best singleton conjectures do not always produce the best pair of conjectures, even when they retain solutions in combination. A better heuristic would need some notion of complementary conjectures. Second, far more combinations are pruned from the search space if failing sets are detected early. Consider the lattice of conjecture sets shown in Figure 7. If conjecture C and D fail to retain solutions when used in combination so will {A,C,D}, {B,C,D} and {A,B,C,D}. A breadth first traversal would be guaranteed to detected this failure early and would consequently never evaluate these three models. Alternatively a depth first traversal would detect this failure late, and would therefore waste time evaluating the supersets of {C,D}, all of which fail to retain solutions.

6 Discussion: Generating Streamliners in Practice

In this section, we consider the process of generating and selecting streamliners when presented with a new problem class of interest. Our methodology is a close analogue of that adopted by human experts in manual streamliner generation. Given an Essence specification of the problem class, we begin by identifying suitable instances with which to evaluate candidate streamliners. These instances must be satisfiable and solvable in reasonable time so that they can be used in the evaluation of a large set of candidate streamlined specifications. This set of instances can be selected manually, or generated automatically by attempting

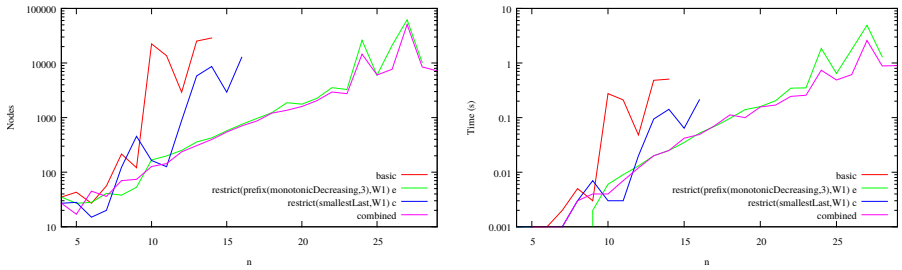


Fig. 9: Combining singleton conjectures to produce a more effective streamlined model for Graceful Helm Graphs. The parameter `n` is the size of the wheel. The first conjecture requires that the differences between the labels of the vertices on the inner loop are in decreasing order, except the last 3. The second requires that the smallest label occurring on the inner loop is the last vertex.

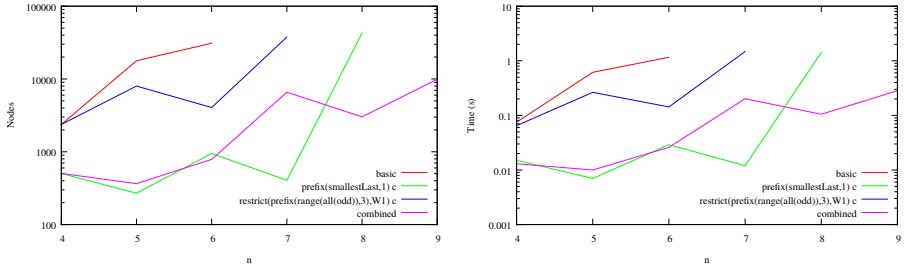


Fig. 10: Combining singleton conjectures to produce a more effective streamlined model for Graceful Double Wheel Graphs. The parameter `n` defines the size of single wheel. The first conjecture requires that the last vertex in the outer loop of the graph takes the largest value. The second requires that the difference between adjacent vertices on the inner loop are odd numbers, except the last 3.

to solve candidates using the basic specification — satisfiable instances solved within a budget are kept for streamliner evaluation.

Streamliners are then generated, combined and evaluated against the set of test instances, as described in Sections 4 and 5. This is a costly process, in the same way that a considerable effort is expended by human experts in manual streamliner generation. However, streamliners are generated for use over the entire problem class. Under the assumption that our problem class has infinitely many elements, the cost of streamliner discovery is amortised over all instances not used in the streamliner evaluation process and becomes negligible.

7 Conclusion

Streamliner generation has been the exclusive province of human experts, requiring substantial effort in examining the solutions to instances of a problem class,

manually forming conjectures as to good streamliners, and then testing their efficacy in practice. In this paper we have demonstrated for the first time the automated generation of effective streamliners, achieved through the exploitation of the structure present in abstract constraint specifications written in Essence. In future work we will expand our set of streamliner generation methods and explore streamliner generation in further, more complex problem classes.

Acknowledgements This work is supported by UK EPSRC grant EP/K015745/1. We thank Ian Gent, Chris Jefferson and Peter Nightingale for helpful comments.

A Problem Descriptions

Van Der Waerden Numbers Van Der Waerden's theorem states that given any positive integers k and l, there exists a positive integer n such that for any partition of the integers 1 to n into k parts at least one of the parts contains an arithmetic sequence of length l. The Van Der Waerden number, $W(k, l)$, is the lowest such n [24]. The Essence specification studied here describes a certificate that the given $n \neq W(k, l)$.

Schur Numbers Given a positive integer r, there exists a positive integer s such that for any partition of the integers 1 to s at least one part is not sum free. Alternatively at least one part is a super set of $\{x, y, z\}$ such that $x + y = z$. Schur's number, $S(r)$, is the smallest such s [22]. The Essence specification studied here describes a certificate that the given $s \neq S(r)$

Graceful Graphs Given a graph with n edges a graceful labelling assigns each node in the graph a label between 0 and n such that no label is used more than once and that every pair of adjacent nodes can be uniquely identified by the absolute difference of their labels. A graceful graph is a graph that permits a graceful labelling [21]. Several classes of graph have been investigate in this context including wheels [9], double wheels [15], helms [3] and gears [17].

Quasigroup Existsence Given a positive integer n, does there exist a quasigroup (latin square) of size n such that an additional side constraint is met. These side constraints are: QGE3 - $\forall a, b \in g \quad (a \cdot b) \cdot (b \cdot a) = a$, QGE4 - $\forall a, b \in g \quad (b \cdot a) \cdot (a \cdot b) = a$, QGE5 - $\forall a, b \in g \quad ((b \cdot a) \cdot b) \cdot b = a$, QGE6 - $\forall a, b \in g \quad (a \cdot b) \cdot b = a \cdot (a \cdot b)$, QGE7 - $\forall a, b \in g \quad (b \cdot a) \cdot b = a \cdot (b \cdot a)$ [23].

Equidistant Frequency Permutation Arrays Given v, q, λ and d, construct a set of v codewords such that each code word is of length $q \cdot \lambda$ and contains λ occurrence of each symbol in the set $\{1, 2, \ldots, q\}$. Each pair of code words must be of hamming distance d [12].

Langford's Problem Given any positive integer n, arrange copies of the numbers between 1 and n such that for all k in $\{1 \ldots n\}$ there are k digits between occurrences of k [14].

References

1. Akgun, O., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure, vol. 8124 LNCS, pp. 107–116 (2013)
2. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAAI-11: Twenty-Fifth Conference on Artificial Intelligence (2011)
3. Ayel, J., Favaron, O.: Helms are graceful. Progress in Graph Theory (Waterloo, Ont., 1982), Academic Press, Toronto, Ont pp. 89–92 (1984)
4. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: ECAI. vol. 141, pp. 73–77 (2006)
5. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: Walsh, T. (ed.) Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming. pp. 575–579 (2001)
6. Colton, S.: Automated Theory Formation in Pure Mathematics. Ph.D. thesis, University of Edinburgh (2001)
7. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: Proc. of the IJCAI 2005. pp. 109–116 (2005)
8. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3) pp. 268–306 (2008), http://dx.doi.org/10.1007/s10601-008-9047-y
9. Frucht, R.: Graceful numbering of wheels and related graphs. Annals of the New York Academy of Sciences 319(1), 219–229 (1979)
10. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. vol. 141, pp. 98–102 (2006)
11. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. In: Principles and Practice of Constraint Programming - CP 2004, pp. 274–289. Springer (2004)
12. Huczynska, S., McKay, P., Miguel, I., Nightingale, P.: Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In: Principles and Practice of Constraint Programming-CP 2009, pp. 50–64. Springer (2009)
13. Kouril, M., Franco, J.: Resolution tunnels for improved sat solver performance. In: Theory and Applications of Satisfiability Testing. pp. 143–157. Springer (2005)
14. Langford, C.D.: Problem. The Mathematical Gazette pp. 287–287 (1958)
15. Le Bras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence. pp. 587–593. AAAI Press (2013)
16. Le Bras, R., Gomes, C.P., Selman, B.: On the erdos discrepancy problem. In: Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. vol. 8656, p. 440. Springer (2014)
17. Ma, K., Feng, C.: On the gracefulness of gear graphs. Math. Practice Theory 4, 72–73 (1984)
18. 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) (2008), http://dx.doi.org/10.1007/s10601-008-9041-4
19. Nightingale, P., Akgun, O., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In: Principles and Practice of Constraint Programming - CP 2014. Springer (2014)

20. Prestwich, S.: CSPLib problem 028: Balanced incomplete block designs. `http://www.csplib.org/Problems/prob028`
21. Rosa, A.: On certain valuations of the vertices of a graph. In: Theory of Graphs (Internat. Symposium, Rome. pp. 349–355 (1966)
22. Schur, I.: Über die kongruenz $x^m + y^m \equiv z^m \pmod{p}$. Jahresber. Deutsch. Math. Verein 25, 114–117 (1916)
23. Slaney, J., Fujita, M., Stickel, M.: Automated reasoning and exhaustive search: Quasigroup existence problems. Computers & mathematics with applications 29(2), 115–132 (1995)
24. Van der Waerden, B.L.: Beweis einer baudetschen vermutung. Nieuw Arch. Wisk 15(2), 212–216 (1927)