Towards Understanding Differences Between Modelling Pipelines: a Modelers Perspective

Csobán Balogh
University of St Andrews, School of Computer Science, Scotland

Ruth Hoffmann[1]
School of Computer Science, University of St Andrews, Scotland

Joan Espasa
University of St Andrews, School of Computer Science, Scotland

Abstract

In this work we aim to investigate the capabilities of the MiniZinc and Savile Row constraint programming pipelines from the user's perspective. We evaluate their modelling and reformulation capabilities on a selection of six diverse problem classes using the commonly supported Chuffed solver. Our preliminary findings show that both pipelines are very competitive in performance. However, they seem to cater to distinct user preferences. MiniZinc allows better modeler control, and provides a slightly more expressive language due to the facilities for code organization and reusability. Conversely, Savile Row provides a solid set of default settings and a more consistent performance profile.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Modeling methodologies

Keywords and phrases Constraint Modelling, Constraint Pipelines, Constraint Satisfaction Problems

1 Introduction

As with imperative programming languages like C, the process from writing out the problem constraints to arriving at a solution involves a pipeline of translation from a high-level language to a low-level language that can be directly interpreted by the solver. Modelling problems to be efficiently solved is no trivial task, and as a result there are automated modelling assistants who tailor the models to the required input to specific solvers. More concretely, we consider tailoring as the process of translating the constraint model with given input parameters (an instance) to a form readable by a specific constraint solver.

In this work we are going to focus on two well-known pipelines: `MiniZinc` [12] and its homonymous language, and `Savile Row` [14] and its `Essence'` [13] language. Both `MiniZinc` and `Essence'` are high-level constraint modelling languages. The `Savile Row` modelling assistant was developed between the University of York and St Andrews. `MiniZinc` is an open-source language, developed at Monash University in collaboration with Data61 Decision Sciences and the University of Melbourne. Similarly, it has its own modelling assistant to tailor the input for the solvers. The following steps describe the usual workflow for a problem to be able to be solved using these two pipelines: **1.** Using a high-level constraint modelling language, a problem is modelled. **2.** The constraints are merged with the instance data and tailored to be fed as input to a specified constraint solver. **3.** The constraint solver reads the low-level description and searches for solutions. **4.** If a solution is found, the solution is then translated back to a user-readable form. We have decided to focus on `Chuffed` [5], as the solver is used by both pipelines, given its robustness, performance and for the maturity of its support by both pipelines.

[1] corresponding author

We selected a set of 6 diverse problems from `CSPLib` [7] and the Minizinc Challenges [17]. While the benchmarks for these problems already existed encoded in either `MiniZinc` or `Essence'`, no direct detailed comparison has been made between the two due to models not being available in both languages. In addition, to our knowledge our work is unique in considering the full pipeline: from modelling, translation and optimisation, to the effects upon the solver. Further, while two models for the same problem may arrive at the same solution, variations in a model, such as a different viewpoint, can result in great variations in solving time [9]. Our objective is to manually create models as similar as possible to try to isolate the pipeline effects as much as possible.

In summary, our contributions are as follows: (i) new models in both `Essence'` and `MiniZinc` of existing problems which allow to directly compare the considered pipelines, (ii) a discussion on how both languages compare when modelling the selected problems, (iii) an empirical evaluation of the pipelines performance over 3 different optimisation levels.

2 The Models

We now present the models for the categories and problems which we chose for this work: mathematical objects (Quasigroup Completion [15]), packing problems (Wordpress [4]), scheduling problems (Rotating Rostering [18] and Travelling Tournament Problem with Predefined Values [16]), planning problems (Multi-Skilled Project Scheduling Problem) and routing problems (Capacitated Vehicle Routing Problem with Time Windows). All models can be found at `https://github.com/stacs-cp/modref2024-pipeline-comparison`.

`MiniZinc` has predicates and further rich functionality and syntax that `Essence'` is lacking. However, in all cases an equivalent expression can be created and run in `Essence'` to produce valid solutions. The main difference in both pipelines is the treatment of the search order. `MiniZinc` has an extensive number of search orders it may impose upon the variables, for example searching through an array of variables from the maximum of the domain to the minimum. `Essence'` on the other hand provides similar functionality for the target solver `Minion` with search heuristics, but these fall short in control compared to `MiniZinc`. We have tried to recreate the `MiniZinc` search order through a rough translation by specifying which variables to branch on first, but there is little control in how to perform the search over the variables and their respective domains.

2.1 Quasigroup Completion

In Quasigroup Completion `CSPLib` [15], an order n quasigroup can be viewed as a size n Latin square, where a Latin square is a $n \times n$ table of numbers ranging from 1 up to n inclusive. The table's rows and columns must be distinct in numbers. The Quasigroup Completion problem gives an incomplete quasigroup table and requires the remaining empty cells to be filled in with numbers.

The provided `MiniZinc` model on `CSPLib` uses an explicit representation, making use of the `allDifferent` constraint to ensure columns and rows are distinct. The `allDifferent` constraint takes in an array, and imposes that every element appears at most once in the array. Given the simplicity of the problem and its reliance upon `allDifferent` through the explicit model, an equivalent occurrence model was created (for both pipelines). The occurrence model represents numbers through an array of zeros and ones [8].

To impose `allDifferent` upon an occurrence representation, either `allDifferent_except` or `sum` could be used. Rather than compare slight variations of `allDifferent`, the interest is in comparing the difference between using a simple operator such as `sum`, and a powerful optimised propagator such as `allDifferent`.

There does not exist an `Essence'` model for Quasigroup Completion on `CSPLib` , so one needed to be created. The explicit model follows the `MiniZinc` model exactly, except `Essence'` has a nice spread operator for slicing columns and rows from matrices instead of list comprehensions. In `Essence'` we represent the occurrence model, by finding a three dimensional $0, 1$ matrix and using a `sum` constraint.

2.2 Wordpress Application Deployment in the Cloud

Wordpress Application Deployment in the Cloud [4] was used in the 2022 `MiniZinc` challenge [10] with minor adaptations. Wordpress is an application used to create websites and needs to be deployed to the cloud with a set of hardware and virtual machines (VMs). The goal is to deploy several Wordpress instances while minimising the cost of hardware, VM configurations and providers to deploy the instances into the cloud. The Wordpress problem is an example of a bin packing problem, a common problem in combinatorial optimization ranging over many applications such as loading trucks or graphics card resource allocation.

The `MiniZinc` model used is taken from the `MiniZinc` challenge 2022 [10]. The original `MiniZinc` model [6], which the `MiniZinc` challenge model is based on, included variables not used within the input files given. More concretely, `FVariables` and `FVInstances` are Fixed Variables and Fixed Variable Instances respectively and were used to determine the effects of fixing certain variables otherwise left to be optimized and found. The paper also originally left the number of VMs as a decision variable, however the `MiniZinc` challenge model has an input parameter for the upper bound on the number of VMs to use. This restricts the search space on the number of VMs and focusing on the objective of minimising the cost.

The `Essence'` model is a one-to-one translation of the `MiniZinc` model. Note the `MiniZinc` model does not break symmetry upon the distribution of hardware on the VMs. The hardware deployed on a VM can be swapped to any other VM for an equivalent solution, resulting in the symmetry of the problem. The original model [6] does break the symmetry of the problem, using multiple different methods such as pricing and the lexicographical assignment of hardware on machines among others. This change from the paper to the `MiniZinc` challenge model might have been done to test a solvers ability to recognize symmetry and break it.

While the non-symmetric version of the problem is of interest following the `MiniZinc` challenge to compare to, the symmetric model is more attuned to how a modeller would write this given problem. We created a version of the problem in `Essence'` that breaks the aforementioned symmetries. To remain fair, an equivalent symmetry breaking version of the `MiniZinc` model was created to compare the symmetry breaking version of the `Essence'` model. Both the symmetry-breaking and non symmetry-breaking models will be compared.

2.3 Rotating Rostering Problem

The Rotating Rostering Problem [18] generalises many real life rostering problems, such as nurse rostering. The goal is to find a satisfiable assignment of shifts for each day to fulfill the shifts requirements and avoid conflicts. A shift type may either be a day off, a morning shift, a late shift, or a night shift. The shifts are ordered such that the following constraints are satisfied: **1.** The number of staff required for a day is satisfied as needed. **2.** There is a min and max number of consecutive shift assignments for the same shift type. **3.** The shift ordering is forward rotating; a shift must be proceeded by a larger or equal shift type, or a rest day. **4.** Weekends (Saturday Sunday) have the same shift type. **5.** At least two days must be rest days every 14 days.

When converting the `MiniZinc` model over to `Essence'`, we identified what we believe is an overlooked edge case. Incorrect solutions rarely appeared, but surfaced when running generated instance solutions through a solution checker. The issue related to the minimum number of shift assignments mandated by the constraints. As part of the original constraints, if the shift type of one day is different to the next day, the following days must meet the minimum number of consecutive shifts. This works well, until the edge case of the very first shift forgoing the minimum number of consecutive shifts, as the constraints ensure the **following** days meet the minimum number of consecutive shifts as seen in Listing 1. To fix this, we simply an extra constraint from the very first shift as seen in Listing 2.

Listing 1 `MiniZinc` minimum number of consecutive shifts constraint

```
constraint forall(day in 1..numberOfDays - s_min) (
   plan1d[day] != plan1d[day+1] -> all_equal(plan1d[day+1..day+s_min]));
```

Listing 2 `MiniZinc` minimum number of consecutive shifts additional constraint

```
constraint(all_equal(plan1d[1..s_min]));
```

2.4 Traveling Tournament Problem with Predefined Venues

The Traveling Tournament Problem with Predefined Venues (TTPPV) [16], is a specialisation of the Traveling Tournament Problem. A set of teams playing in a tournament is organized as a simple round robin schedule, with each game playing at different venues. The objective is to minimize the distance travelled by the teams between different venues.

The difference to the Traveling Tournament Problem is that in TTPPV the venues of each game are predefined. This means if team a plays against team b, the venue is predefined as being at either a's home or b's home. With the predefined venues, the problem lies in the scheduling of the games and minimizing the sum of the traveling distances of the teams. The problem is further specialised by using circular distances between the venues for simplicity.

The `regular` predicate is used within the `MiniZinc` model to assert there are at most 3 consecutive home games and at most 3 consecutive away games. More generally, the `regular` predicate asserts that a sequence of variables take a value from a finite automaton, where the automaton in the `MiniZinc` model asserts at most two consecutive away or home games. When translating the `regular` predicate to `Essence'`, we use a `forAll` statement, checking that there are not four consecutive assignments.

2.5 Capacitated Vehicle Routing problem with Time Windows, Service Times and Pickup and Deliveries

The Capacitated Vehicle Routing problem with Time Windows, Service Times and Pickup and Deliveries (CVRPTW) is an example of a routing problem which specialises the Capacitated Vehicle Routing Problem [19] and was sourced from the 2022 `MiniZinc` challenge [17]. It is defined as follows: there are several vehicles with a given capacity for goods, and there are a number of pickup and drop-off locations of customers to deliver to. These pickup and delivery locations have an associated demand for the goods the vehicles need to pick up or deliver. As an added constraint, there are specified time windows for the deliveries to each customer such that the delivery truck must arrive and leave within this time window from the delivery location. The route chosen may not take any sub-tours for any route it takes.

As part of the `MiniZinc` model, the `circuit` predicate is used to ensure the vehicle delivery routes do not take sub-tours in their route and visits each location uniquely for optimisation. A circuit is such that the cell value of an array points to the index of the next number, and this forms a circuit that continues around. For the translation to `Essence'`, we used the decomposition of the `circuit` predicate from the `MiniZinc` library [2].

To create the equivalent expression of circuit in `Essence'`, a new variable is introduced to determine and constrain the ordering of values to form the circuit.The `Essence'` equivalent of circuit for the decision variable `successor` in the model is as follows in Listing 3.

Listing 3 `Essence'` `circuit` predicate equivalent

```
allDiff(successor),
forAll i : NODES . successor[i] != i,
allDiff(successorOrder),
successorOrder[1] = 1,
forAll i : NODES .
    (successorOrder[i] = maxNodes -> successorOrder[successor[i]] = 1) /\
    (successorOrder[i] != maxNodes -> successorOrder[successor[i]] =
        successorOrder[i] + 1)
```

2.6 Multi-Skilled Project Scheduling Problem

The Multi-Skilled Project Scheduling Problem (MSPSP) is a variation on the basic resource-constraint project scheduling problem [1] used in the 2012 Minizinc Challenge. In this problem there are a series of workers, with each worker having a specific skill set. There are several activities with an associated skill requiring completion to finish the project, and the overall goal is to minimize the project time.

The `MiniZinc` formulation uses set variables, where `Essence'` (unlike `Essence` [3]) lacks modelling support for them. To overcome this limitation, the sets from `MiniZinc` were translated into the occurrence representation of the numbers. This allowed for each matrix to be equivalent in size to satisfy the `Essence'` language limitations. A disadvantage of using this method is that by using the occurrence representation the parameter files become larger.

The `MiniZinc` model also makes use of `letting` to create variables within constraints, but `Essence'` cannot do the same. As a result, an equivalent expression is created. On Line 4 of Listing 4, a new Boolean variable is introduced into the scope of the constraint. This variable acts like a normal decision variable, with the goal of assigning a satisfiable value. In Line 5 and Line 6 the Boolean variable `before` in combination with an implication ensures at least one of the expressions following the implication is true. This can be compactly expressed in `Essence'` by using an *or*, as shown in Listing 5 on Line 5.

Listing 4 Usage of `MiniZinc` `letting` in MSPSP

```
1 constraint
2    forall (i, j in Tasks where i < j /\ not(j in suc[i]) /\ not(i in suc[j]))(
3        if exists( k in Skills )( rr[k,i] + rr[k,j] > rc[k] ) then
4            let { var bool: before } in (
```

[2] `fzn_circuit.mzn` in [11]

```
              (before -> s[i] + d[i] <= s[j])
            /\  (not(before) -> s[j] + d[j] <= s[i]) )
        else true endif);
```

Listing 5 Essence' letting equivalent to Listing 4

```
forAll i : Tasks . forAll j : Tasks .
    (i < j /\ suc[i,j] = 0 /\ suc[j,i] = 0) ->
        ((exists k : Skills .
            rr[k,i] + rr[k,j] > rc[k]) ->
                ((s[i] + d[i] <= s[j]) \/ (s[j] + d[j] <= s[i]))),
```

The MiniZinc model (Listing 6) has a series of further lettings: WTasks and TWorkers. WTasks and TWorkers are sets, where WTasks is the set of tasks where a skill for that task exists, and TWorkers is the set of workers who have an existing skill required. To create an equivalent in Essence' , a single variable is introduced, TWorkers, encompassing WTasks and TWorkers together in a 2d matrix. This is expressed in Listing 7.

Listing 6 Additional MiniZinc lettings in MSPSP

```
let { set of int: WTasks =
        { i | i in Tasks where exists(k in has_skills[j])(rr[k, i] > 0) }
} in...
let { set of int: TWorkers =
        { j | j in Workers where exists(k in has_skills[j])(rr[k, i] > 0) }
} in...
```

Listing 7 Essence' equivalent to Listing 6

```
forAll i : Tasks . forAll j : Workers .
    TWorkers[j, i] = 1 <->
        exists k : Skills . has_skills[j, k] = 1 /\ rr[k,i] > 0,
```

TWorkers is then leveraged in all further constraints equivalent to the lettings of TWorkers and WTasks as seen in Listing 8. Listing 8 constraints the number of workers with a set skill working upon a task to satisfy the requirements. Using the TWorkers variable created in Listing 7, the equivalent of the MiniZinc Listing 8 is created for Essence' in Listing 9.

Listing 8 MiniZinc letting over TWorkers

```
constraint forall ( i in Tasks ) (
    let {
        set of int: TWorkers =
            { j | j in Workers where exists(k in has_skills[j])(rr[k, i] > 0) }
    } in (
        forall ( k in Skills where rr[k, i] > 0 )
            (sum(j in TWorkers where k in has_skills[j])(
            bool2int(w[j, i])) >= rr[k, i])
    /\ forall ( j in Workers where not(j in TWorkers) )
        (w[j, i] = false)));
```

Listing 9 Essence' equivalent to Listing 8

```
forAll i : Tasks . forAll k : Skills .
      rr[k, i] > 0 ->
         sum([w[j,i] /\ TWorkers[j,i] = 1 /\ has_skills[j,k] = 1 | j : Workers])
             >= rr[k, i],
forAll i : Tasks . forAll j : Workers .
      TWorkers[j, i] = 0 -> w[j,i] = false,
```

The `cumulative` predicate is used to determine if the cumulative resource usage is within bounds. That is, a set of tasks with start times, durations, and resource requirements, never exceed the global resource bound at any time. The `cumulative` predicate is a common predicate used in scheduling problems and is therefore optimized in most solvers such as `Chuffed`. In the translation to `Essence'` we used the default `MiniZinc` decomposition of `cumulative`[3]. The `cumulative` predicate is used twice in the MSPSP `MiniZinc` model, both with `letting` statements.The first `cumulative` imposes that at least one worker fulfills the task assignment while respecting the duration and timings. The second `cumulative` ensures the resources requirements is exceeded or equaled by workers while respecting durations and orderings. The equivalent `Essence'` is created in Listing 10.

Listing 10 `Essence'` equivalent of `cumulative` in MSPSP

```
forAll work : Workers .
   sum(TWorkers[work,..]) > 1 ->
      (forAll j : Tasks .
         1 >= sum([(TWorkers[work, j] = 1) /\ (TWorkers[work, i] = 1) /\
            (s[i] <= s[j]) /\ (s[j] < (s[i] + d[i]))
            /\  w[work,i] | i : Tasks])),
forAll k : Skills .
   (sum([rr[k,i] > 0 | i : Tasks]) > 1 /\
    sum([rr[k,i] | i : Tasks, rr[k,i] > 0]) > rc[k]) ->
      forAll j : Tasks .
         rr[k,j] > 0 ->
            rc[k] >= sum([(s[i] <= s[j] /\ s[j] < (s[i] + d[i]))*rr[k,i] | i :
                Tasks, rr[k,i] > 0]),
```

3 Experiments

Our experiments aim to identify performance differences between `Savile Row` 1.9.1 and `MiniZinc` 2.7.5. The experiments were run using 3 different optimisation levels that the respective developers offer for each pipeline. That is, no optimisation (O0S0 in `Savile Row`, O0 in `MiniZinc`), intermediate optimisation (O2S1 in `Savile Row`, O1 in `MiniZinc`) and full optimisations (O3S2 in `Savile Row`, O5 in `MiniZinc`). Note that these will differ between pipelines, in particular due to `Savile Row` allowing control over the amount of symmetry breaking constraints introduced during tailoring, something that `MiniZinc` does not enable.

Certain problems were lacking in the number of instances or in their variety. To compensate, additional instances were generated through a combination of Python scripts or parameterised generators as constraint models, similarly to previous approaches [2].

Timing information is presented as the quotient of `Essence'` over `MiniZinc`. That is, a

[3] `fzn_cumulative_task` of `fzn_cumulative.mzn` in [11]

number > 1 suggests `MiniZinc` was faster, while a number < 1 suggests `Essence'` was faster. These timings use the geometric mean, which uses the product rather than the sum, giving a better indicator of the central tendency of runs. In Table 1 we can see that both pipelines produce very similar and consistent results with respect to solving a given problem. Both pipelines solve all instances for the Rostering and Scheduling problems, and neither finds any solutions to the vehicle routing problems (within the set timeout). When the problems get harder (or more varied), such as the Quasigroup problems, modelling in `Savile Row` seems to be more consistent between the model representations and different optimisations levels. In the other problem groups both pipelines can be considered equal.

From a timing perspective, `MiniZinc` clearly outperforms `Savile Row` in MSPSP and Rostering (the problem classes where both solvers find all solutions). Meanwhile, in the two Quasigroup models `Savile Row` performs faster, and is solving more of the instances. We see this as an indicator that, in the given dataset, `Savile Row` performs better on harder instances. We believe that `MiniZinc` outperforms `Savile Row` on easy instances, as `MiniZinc` has a very low initialisation time when compared to `Savile Row` , as it is written in C++. In problems where both pipelines solved some of the problems, the results are inconclusive. Although the Wordpress problem without the explicit symmetry breaking constraints performs better in the highest optimisation level in `MiniZinc`, with the explicit symmetry breaking constraints `Savile Row` performs better the higher the optimisation levels.

		`Essence'`			`MiniZinc`			Timing Ratio		
Problem	#	O0S0	O2S1	O3S2	O0	O1	O5	$\frac{O0S0}{O0}$	$\frac{O2S1}{O1}$	$\frac{O3S2}{O5}$
Quasigroup	43	41	42	41	40	39	40	0.08	0.5	0.6
Quasigroup Occ.	43	41	41	42	32	37	38	0.12	0.08	0.38
Wordpress	9	6	6	6	6	6	6	1.54	1.83	5.29
Wordpress Symm.	9	4	4	6	4	4	4	1.47	1.36	0.49
TTPPV	20	3	3	3	3	3	3	0.99	1.35	1.98
MSPSP	6	6	6	6	6	6	6	138.48	88.57	612.61
CVRPTW	5	0	0	0	0	0	0	1.0	1.0	1.0
Rostering	7	7	7	7	7	7	7	29.5	15.15	77.7

Table 1 Columns `Essence'` and `MiniZinc` show the number of solved instances per problem, split between the 3 considered optimisation levels. Timing ratios show the ratio between `Essence'` and `MiniZinc` options, where > 1 denotes `MiniZinc` was faster and < 1 otherwise.

4 Conclusions and Further Work

These initial findings seem to suggest that Minizinc might be better suited for scenarios where an expert modeler can leverage the capabilities of the pipeline and where code maintainability are crucial. Conversely, Savilerow strong reformulation capabilities and good default settings would be the preferred choice for tackling complex problems where consistent performance is paramount. To solidify these findings, a wider selection of both solvers and problems have to be considered.

References

1 Rina Agarwal, Manoj K Tiwari, and Sanat K Mukherjee. Artificial immune system based approach for solving resource constraint project scheduling problem. *The International Journal*

of Advanced Manufacturing Technology, 34(5):584–593, 2007.

2 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, and Christopher Stone. Instance generation via generator instances. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, pages 3–19, Cham, 2019. Springer International Publishing.

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. *Artif. Intell.*, 310:103751, 2022. doi:10.1016/J.ARTINT.2022.103751.

4 David Bogdan-Nicolae. CSPLib problem 090: Wordpress application deployment in the cloud. http://www.csplib.org/Problems/prob090.

5 Geoffrey Chu, Peter J Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2018. URL: https://github.com/chuffed/chuffed.

6 Mădălina Eraşcu, Flavia Micota, and Daniela Zaharie. Scalable optimal deployment in the cloud of component-based applications using optimization modulo theory, mathematical programming and symmetry breaking. *Journal of Logical and Algebraic Methods in Programming*, 121:100664, 2021.

7 Ian P Gent and Toby Walsh. Csplib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.

8 Christopher Jefferson and Alan M Frisch. On the effectiveness of set and multiset representations in constraint programming. *Modelling and Reformulating Constraint Satisfaction Problems*, page 125, 2004.

9 Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–32, 1992.

10 MiniZinc. Minizinc challenge 2022 results, 2022. URL: https://www.minizinc.org/challenge2022/results2022.html.

11 MiniZinc Developers. MiniZinc Standard Library. https://github.com/MiniZinc/libminizinc/tree/master/share/minizinc/std.

12 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

13 Peter Nightingale. Savile row manual. *CoRR*, abs/2201.03472, 2022. URL: https://arxiv.org/abs/2201.03472, arXiv:2201.03472.

14 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. URL: https://www.sciencedirect.com/science/article/pii/S0004370217300747, doi:https://doi.org/10.1016/j.artint.2017.07.001.

15 Gilles Pesant. CSPLib problem 067: Quasigroup completion. http://www.csplib.org/Problems/prob067.

16 Gilles Pesant. CSPLib problem 068: Traveling tournament problem with predefined venues (ttppv). http://www.csplib.org/Problems/prob068.

17 Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014.

18 Petra Hofstedt Sven Löffler, Ilja Becker. CSPLib problem 087: Rotating rostering problem. http://www.csplib.org/Problems/prob087.

19 Özgür Akgün. CSPLib problem 086: Capacitated vehicle routing problem. http://www.csplib.org/Problems/prob086.