

# SAVILE ROW Manual

Peter Nightingale

## 1 Introduction

SAVILE ROW is a constraint modelling tool. It provides a high-level language for the user to specify their constraint problem, and automatically translates that language to the input language of a constraint solver. SAVILE ROW is a research tool, designed to enable research into reformulation of constraint models, therefore it is designed for flexibility. It is designed to be easy to add new transformation rules and develop new translation pipelines.

This manual is for version 1.8.0 of SAVILE ROW.

This manual covers the basics of using SAVILE ROW. It does not cover adding new rules or translation pipelines. SAVILE ROW reads the ESSENCE PRIME modelling language, which is described in the ESSENCE PRIME description. SAVILE ROW converts constraint models expressed in ESSENCE PRIME into the solver input format, in a process that has some similarities to compiling a high-level programming language. Like a compiler, SAVILE ROW applies some optimisations to the model (the most basic being partial evaluation and common subexpression elimination).

### 1.1 Problem Classes and Instances

The distinction between *problem classes* and *problem instances* will be important in this manual. It is easiest to start with an example. Sudoku in general is a problem class, and a particular Sudoku puzzle is an instance of that class. The problem class is described by writing down the rules of Sudoku, i.e. that we have a  $9 \times 9$  grid and that each row, column and subsquare must contain all values 1..9. A particular instance is described by taking the class and adding the clues, i.e. filling in some of the squares. The set of clues is a *parameter* of the class – in this case the only parameter. Adding the parameters to a class to make a problem instance is called *instantiation*.

In typical use, SAVILE ROW will read a problem class file and also a parameter file, both written in ESSENCE PRIME. It will instantiate the problem class and unroll all quantifiers and matrix comprehensions, then perform reformulations and flattening of nested expressions before producing output for a constraint solver.

### 1.2 Translation Pipelines and Solver Backends

The current version of SAVILE ROW has two translation pipelines. The first works on problem instances only. Every parameter of the problem class must be given a value in the parameter file. Output produced by this pipeline may be entirely flattened or it may contain some nested expressions depending on the output language.

- Minion – Output is produced in the Minion 3 language for Minion 1.8 or later. The model produced is not entirely flat, it makes use of nested `watched-or` and `watched-and` constraints.
- Gecode – Output is produced for Gecode in the entirely flat, instance-level language FlatZinc for use by the `fzn-gecode` tool.
- Chuffed – Output is produced for the Chuffed solver in the FlatZinc language. The Chuffed and Gecode backends produce very similar FlatZinc.
- SAT – Output is produced in DIMACS format for use with any SAT solver.
- MaxSAT – Produces Weighted Partial MaxSAT for use with MaxSAT solvers such as Open-WBO and MaxHS.

- MiniZinc – Output is produced in an instance-level, almost flat subset of the MiniZinc modelling language. The MiniZinc output follows the FlatZinc outputs as closely as possible. The MiniZinc toolchain can then be used to run a variety of solvers.

The Minion, Gecode, Chuffed, SAT, and MaxSAT backends are able to run the solver and parse the solution. For these backends, SAVILE ROW will produce a solution file (in ESSENCE PRIME) and a file with some statistics. The SAT backend makes multiple calls to solve optimisation problems (using a bisecting strategy by default). All constraints (i.e. the entire ESSENCE PRIME language) is supported with each backend. For the bundled solvers (Minion, Glucose (SAT), Chuffed), paths are set so that SAVILE ROW always uses the bundled solver by default.

The second translation pipeline does not instantiate the model, instead it performs class-level reformulation and flattening. This pipeline is experimental and incomplete. There is one backend that produces Dominion Input Language for the Dominion solver synthesiser.

MINION is well supported by SAVILE ROW so we will use MINION as the reference in this document. MINION is a fast scalable constraint solver. However, modelling problems directly in MINION's input language is time-consuming and tedious because of its primitive structure (it can be compared to writing a complex program in assembly language).

## 2 Installing and Running SAVILE ROW

SAVILE ROW is distributed as an archive with the following contents:

- The Java source code (in `src`) licensed with GPL 3.
- The compiled classes in a JAR file named `savilerow.jar`.
- Executable script `savilerow` for running SAVILE ROW.
- This document and the ESSENCE PRIME description, in `doc`.
- Various example ESSENCE PRIME files and parameter files, in `examples`.
- Required Java libraries in `lib`
- Executables for backend solvers Minion, Glucose, and Chuffed, and the graph automorphism solver Ferret in `bin`

Archives are provided for Linux and Mac, and the Linux archive works well on Windows 10 in the Ubuntu app. The two distributions are largely the same, with the only difference being the packaged executables in `bin`.

A recent version of Java is required on all platforms. The included JAR file was compiled with Oracle JDK 1.8.

### 2.1 Running SAVILE ROW on Linux and Mac

Download the appropriate archive and unpack it somewhere convenient. Open a terminal and navigate to the SAVILE ROW directory. Use the script named `savilerow`. One of the simplest ways of running SAVILE ROW is given below.

```
./savilerow problemFile.eprime parameterFile.param
```

The first argument is the problem class file. This is a plain text file containing the constraint problem, expressed in the ESSENCE PRIME language.

The second argument (`parameterFile.param`) is the parameter file (again in the ESSENCE PRIME language). This contains the data for the problem instance. The parameter file can be omitted if the problem has no parameters (no given statements in the preamble).

## 2.2 Running SAVILE ROW on Windows

The current (1.8.0) release of SAVILE ROW does not have an archive for Windows specifically, however the Linux archive works well in the Ubuntu app on Windows 10. To install, first enable the Windows Subsystem for Linux. Second, install the Ubuntu app and update packages in it (for example with `sudo apt update; sudo apt upgrade`). Third, install Java (for example, `sudo apt install default-jre`). Then follow the instructions for installing and running on Linux.

## 2.3 Solution Files

For the Minion, SAT, MaxSAT, Gecode and Chuffed backends, SAVILE ROW is able to run the solver and parse the solution (or set of solutions) produced by the solver. These solutions are translated back into ESSENCE PRIME. For each `find` statement in the model file (i.e. each statement that declares decision variables), the solution file contains a matching `letting` statement. For example, if the model file contains the following `find` statement:

```
find M : matrix indexed by [int(1..2), int(1..2)] of int(1..5)
```

The solution file could contain the following `letting` statement.

```
letting M = [[2,3],
             [1,2]]
```

## 2.4 File Names

The input files for SAVILE ROW typically have the extension `.eprime`, `.param`, or `.eprime-param`. These extensions allow SAVILE ROW to identify the model and parameter file on the command line. If these files have a different extension (or no extension) then SAVILE ROW must be called in a slightly different way:

```
./savilerow -in-eprime problemFile -in-param parameterFile
```

Given input file names `<problemFile>` and `<parameterFile>`, output files have the following names by default.

- For Minion output, `<parameterFile>.minion` (or if there is no parameter file, `<problemFile>.minion`).
- For Gecode or Chuffed output, `<parameterFile>.fzn` (or if there is no parameter file, `<problemFile>.fzn`).
- For SAT or MaxSAT output, `<parameterFile>.dimacs` (or if there is no parameter file, `<problemFile>.dimacs`).
- For Minizinc output, `<parameterFile>.mzn` (or if there is no parameter file, `<problemFile>.mzn`).
- For Dominion output, there is typically no parameter file so the output file is named after the problem file. If the problem file ends with `.eprime`, then the extension is removed. The extension `.dominion` is added.

When SAVILE ROW parses a single solution from the output of a solver, it produces a file named `<parameterFile>.solution` (or if there is no parameter file, `<problemFile>.solution`). When there are multiple solutions (e.g. when using the `-all-solutions` flag) the solution files are numbered (for example, `nurses.param.solution.000001` to `nurses.param.solution.000871`).

If SAVILE ROW runs Minion or a SAT solver it produces a file `<parameterFile>.info` (or if there is no parameter file, `<problemFile>.info`) containing solver statistics.

Finally, a file named `<parameterFile>.aux` (or `<problemFile>.aux`) is also created. This contains the symbol table and is read if SAVILE ROW is called a second time to parse a solution.

## 2.5 Command Line Options

### 2.5.1 Mode

SAVILE ROW has two modes of operation, Normal and ReadSolution. Normal is the default, and in this mode SAVILE ROW reads an ESSENCE PRIME model file and optional parameter file and produces output for some solver. In some cases it will also run a solver and parse the solution(s), producing ESSENCE PRIME solution files.

In ReadSolution mode, SAVILE ROW reads a Minion solution table file. The solution table file is created by running Minion with its `-solsout` flag. The solution or solutions saved by Minion are converted to ESSENCE PRIME solution files. In the process, any decision variables that were removed by SAVILE ROW are restored, and all auxiliary variables that were introduced by SAVILE ROW are removed.

The mode is specified as follows.

```
-mode [Normal | ReadSolution]
```

### 2.5.2 Specifying input files

The options `-in-eprime` and `-in-param` specify the input files as in the example below.

```
savilerow -in-eprime sonet.eprime -in-param sonet1.param
```

These flags may be omitted for any filename that ends with `.eprime`, `.param`, or `.eprime-param`.

The option `-params` may be used to specify the parameters on the command line. For example, suppose the `nurse.eprime` model has two parameters. We can specify them on the command line as follows. The format of the parameter string is identical to the format of a parameter file (where, incidentally, the `language` line is optional and line breaks are not significant).

```
savilerow -in-eprime nurse.eprime -params "letting n_nurses=4 letting Demand=[[1,0,1,0],[0,2,1,0]]"
```

### 2.5.3 Specifying output format

In Normal mode, there are seven output formats as described in Section 1.2. The output format is specified using one of the following command-line options.

```
-minion
-gecode
-chuffed
-sat
-maxsat
-minizinc
-dominion
```

The output filename may be specified as follows. In each case there is a default filename so the flag is optional. Default filenames are described in Section 2.4. The flag `-out-sat` also sets the MaxSAT filename.

```
-out-minion <filename>
-out-gecode <filename>
-out-chuffed <filename>
-out-sat <filename>
-out-minizinc <filename>
-out-dominion <filename>
```

In addition, the file names for solution files, solver statistics files and aux files may be specified as follows. Once again there are default filenames described in Section 2.4.

```
-out-solution <filename>
-out-info <filename>
-out-aux <filename>
```

### 2.5.4 Optimisation Levels

The optimisation levels (`-O0` to `-O3`) provide an easy way to control how much optimisation SAVILE ROW does, without having to switch on or off individual optimisations. The default is `-O2`, which is intended to provide a generally recommended set of optimisations. The rightmost `-O` flag on the command line is the one that takes precedence.

`-O0`

The lowest optimisation level, `-O0`, turns off all optional optimisations. SAVILE ROW will still simplify expressions (including constraints). Any expression containing only constants will be replaced with its value. Some expressions have quite sophisticated simplifiers that will run even at `-O0`. For example, `allDiff([x+y+z, z+y+x, p, q])` would simplify to `false` because the first two expressions in the `allDiff` are symbolically equal after normalisation. `-O0` will do no common subexpression elimination, will not unify equal variables, and will not filter the domains of variables.

`-O1`

`-O1` does optimisations that are very efficient in both space and time. Variables that are equal are unified, and a form of common subexpression elimination is applied (Active CSE, described below).

`-O2`

In addition to the optimisations performed by `-O1`, `-O2` performs filtering of variable domains and aggregation (both of which are described in the following section). `-O2` is the default optimisation level.

`-O3`

In addition to `-O2`, `-O3` enables tabulation (`-tabulate`) and associative-commutative common subexpression elimination (`-ac-cse`) (both described below).

### 2.5.5 Symmetry-Breaking Levels

The options `-S0`, `-S1` and `-S2` control optional optimisations that can change the number of solutions. The default is `-S1` which applies only very simple and fast optimisations.

`-S0`

Preserve the number of solutions, i.e. perform no optimisations that might change the number of solutions.

`-S1`

Remove variables that appear in no constraints, and allow SAVILE ROW to introduce auxiliary variables that are not functionally defined by the primary variables. At present, non-functional auxiliary variables are only used with the SAT and MaxSAT backends.

`-S2`

In addition to `-S1`, apply a graph automorphism solver to find symmetries among the decision variables. The symmetries are then broken using the standard lex-leader method.

## 2.5.6 Translation Options

### Common Subexpression Elimination

SAVILE ROW currently implements four types of common subexpression elimination (CSE). Identical CSE finds and eliminates syntactically identical expressions. This is the simplest form of CSE, however it can be an effective optimisation. Active CSE performs some reformulations (for example applying De Morgan's laws) to reveal expressions that are semantically equivalent but not syntactically identical. Active CSE subsumes Identical CSE. Active CSE is enabled by default as part of `-O2`. Associative-Commutative CSE (AC-CSE) works on the associative and commutative (AC) operators `+`, `*`, `/` (and) and `\` (or). It is able to rearrange the AC expressions to reveal common subexpressions among them. AC-CSE is not enabled by default. It would normally be used in conjunction with Identical or Active CSE. Finally, Active AC-CSE combines one active reformulation (integer negation) with AC-CSE, so for example Active AC-CSE is able to extract  $y - z$  from the three expressions  $x + y - z$ ,  $w - x - y + z$ , and  $10 - y + z$ , even though the sub-expression occurs as  $y - z$  in one and  $-y + z$  in the other two. Active AC-CSE is identical to AC-CSE for And, Or and Product, it differs only on sum.

The following flags control CSE. The first, `-no-cse`, turns off all CSE. The other flags each turn on one type of CSE.

```
-no-cse
-identical-cse
-active-cse
-ac-cse
-active-ac-cse
```

### Variable Deletion

SAVILE ROW can remove a decision variable (either variables declared with `find` or auxiliary variables introduced during tailoring) when the variable is equal to a constant, or equal to another variable. This is often a useful optimisation. It can be enabled using the following flag.

```
-deletevars
```

### Domain Filtering

It can be useful to filter the domains of variables. In SAVILE ROW this is done by running the translation pipeline to completion and producing a Minion file, then running Minion usually with the options `-preprocess SACBounds.limit -outputCompressedDomains`. With these options Minion performs conventional propagation plus SAC on the lower and upper bound of each variable (SACBounds, also known as shaving) with a limit on the number of iterations. The filtered domains are then read back in to SAVILE ROW. The translation process is started again at the beginning. Thus domain filtering can benefit the entire translation process: variables can be removed (with `-deletevars`), constraint expressions can be simplified or even removed (if they are entailed), the number of auxiliary variables may be reduced. In some cases domain filtering can enable another optimisation, for example on the BIBD problem it enables associative-commutative CSE to do some very effective reformulation.

Domain filtering can be used with Minion, Chuffed, Gecode, SAT, MaxSAT and Minizinc output (but Minion is always used to perform the domain filtering regardless of the output solver). It is switched on using the following flag.

```
-reduce-domains
```

Standard domain filtering affects the decision variables defined by `find` statements. Auxiliary variables created by SAVILE ROW are not filtered by default. To enable filtering of both `find` variables and auxiliary variables, use the following flag:

```
-reduce-domains-extend
```

---

## Aggregation of Constraints

Aggregation collects sets of constraints together to form global constraints that typically propagate better in the target solver. At present there are two aggregators for `allDifferent` and `GCC`. `SAVILE ROW` constructs `allDifferent` constraints by collecting `not-equal`, `less-than` and other shorter `allDifferent` constraints. `GCC` is aggregated by collecting `atmost` and `atleast` constraints over the same scope.

`-aggregate`

## Tabulation

Tabulation converts constraint expressions into table constraints to improve propagation. A set of four heuristics are used to identify candidate expressions for tabulation. When tabulating a constraint, the number of tuples is limited to 10,000 to avoid spending excessive time on one constraint. Refer to the CP 2018 paper for more detail [1].

`-tabulate`

## Mappers

The Dominion constraint solver synthesiser has mappers (also known as views) in its input language. By default `SAVILE ROW` will use these whenever possible. The following two flags control the use of mappers – the first switches mappers off completely, and the second allows mappers to be used only where Minion can use them. For example, multiplication-by-a-constant mappers are allowed on variables in a sum constraint, because Minion has weighted sum constraints (which, incidentally, are built from sum constraints using mappers).

`-nomappers`  
`-minionmappers`

## Integer Variable Types

When creating Minion output `SAVILE ROW` will by default use the `DISCRETE` or `BOOL` variable type when the variable has fewer than 10,000 values and `BOUND` otherwise. `DISCRETE` and `BOOL` represent the entire domain and `BOUND` only stores the upper and lower bound, thus propagation may be lost when using `BOUND`. The following flag prevents `SAVILE ROW` using `BOUND` variables.

`-no-bound-vars`

## Remove Redundant Variables

The following flag causes `SAVILE ROW` to remove any decision variables that are not mentioned in a constraint or in the objective function. It is enabled by default as part of `-S1`.

`-remove-redundant-vars`

## Variable Symmetry Breaking

If variable symmetry breaking is enabled, `SAVILE ROW` will apply a graph automorphism solver to find symmetries among the decision variables. Any symmetries discovered will be broken by the standard lex-leader method. It is disabled by default because it can be expensive. Variable symmetry breaking is part of `-S2`.

`-var-sym-breaking`

### 2.5.7 SAT Encoding Options

The SAT backend in general is described in Appendix A. There are several options for encoding sum constraints in SAVILE ROW.

Sum constraints are divided into three groups and there are flags to control the encoding of each group. The simplest group is at-most-one (AMO) and exactly-one (EO) constraints containing only Boolean terms (not just Boolean variables but any expression of type Boolean) and where all coefficients are 1. The encoding of these constraints is controlled by `-sat-amo` flags. Pseudo-Boolean (PB) sums also contain only Boolean terms but can have coefficients other than 1. The encoding of PB constraints is controlled by `-sat-pb` flags. Finally, the encoding of general sums (containing at least one integer term) is controlled by `-sat-sum` flags.

#### AMO Encodings

The default encoding for AMO and EO constraints is Chen's 2-product, it is small and tends to be efficient. Three other encodings are available: the Commander encoding of Klieber and Kwon (with group size 3); the ladder encoding; and an encoding similar to the totalizer (named Tree). The flags to select the encoding are below:

```
-sat-amo-product    (default)
-sat-amo-commander
-sat-amo-ladder
-sat-amo-tree
```

#### PB Encodings

There are five encodings of Pseudo-Boolean constraints. Four of them are based on Bofill et al [4] (with thanks to Jordi Coll for providing the implementations): Multi-valued Decision Diagrams (MDD); the Global Polynomial Watchdog (GPW); the Sequential Weighted Counter (SWC); and the Generalized Generalized Totalizer (GGT). Each of these is implemented exactly as described in [4]. For these four encodings, any pseudo-Boolean equality constraints are decomposed into  $\leq$  and  $\geq$ . Also these four encodings support only top-level constraints (i.e. constraints that are not nested inside a logic expression). The flags are as follows:

```
-sat-pb-mdd
-sat-pb-gpw
-sat-pb-swc
-sat-pb-ggt
```

The fifth encoding (called Tree) is similar to the Generalized Totalizer. It can directly encode sum-equality, and can also encode non-top-level constraints. It is the default encoding and also the fallback for non-top-level constraints when using one of the other four encodings.

```
-sat-pb-tree    (default)
```

The first four encodings (MDD, GPW, SWC, GGT) are able to take advantage of at-most-one (and exactly-one) relations on the Boolean variables to reduce the size of the encoding. All four of these encodings work with automatic AMO and EO detection as described in the CP 2019 paper [2], which uses Minion to find mutexes between variables then constructs maximal cliques of the mutex graph. The flag below is used to switch on the AMO and EO detection. It is often worthwhile for sufficiently challenging problems with PB constraints.

```
-amo-detect
```



## Sum Encodings

The options for encoding general sums (containing at least one integer variable) are the same as for PB constraints. The flags are as follows:

```
-sat-sum-mdd
-sat-sum-gpw
-sat-sum-swc
-sat-sum-ggt
-sat-sum-tree (default)
```

The Tree encoding natively handles integer terms. The other four (MDD, GPW, SWC, GGT) treat integer terms as a group of Boolean terms (e.g.  $1 * (x = 1) + 2 * (x = 2) + \dots$ ) with an exactly-one relation on them. The four encodings (MDD, GPW, SWC, GGT) also can benefit from automatically detected AMO and EO relations with the `-amo-detect` flag.

### 2.5.8 Controlling SAVILE ROW

The following flag specifies a time limit in milliseconds. SAVILE ROW will stop when the time limit is reached, unless it has completed tailoring and is running a backend solver. The time measured is wallclock time not CPU time.

```
-timelimit <time>
```

Time limits may also be passed through to solvers, for example when using Minion as the backend solver, the following flag will limit Minion's run time (specified in seconds).

```
-solver-options "-cpulimit <time>"
```

A similar approach can be used to apply a time limit to Gecode and SAT solvers.

In some cases the SAT or MaxSAT output can be very large. The following option allows the number of clauses to be limited. If the specified number of clauses is reached, SAVILE ROW will delete the partial SAT file and exit.

```
-cnflimit <numclauses>
```

### 2.5.9 Solver Control

The following flag causes SAVILE ROW to run a solver and parse the solutions produced by it. This is currently implemented for Minion, Gecode, Chuffed, SAT and MaxSAT backends.

```
-run-solver
```

The following two flags control the number of solutions. The first causes the solver to search for all solutions (and SAVILE ROW to parse all solutions). The second specifies a required number of solutions.

```
-all-solutions
-num-solutions <n>
```

When parsing solutions the default behaviour is to create one file for each solution. As an alternative, the following flag will send solutions to standard out, separated by lines of 10 minus signs.

```
-solutions-to-stdout
```

The following flag passes through command-line options to the solver. The string would normally be quoted.

```
-solver-options <string>
```

For example when using Minion `-solver-options "-cpulimit <time>"` may be used to impose a time limit, or when using Gecode `-solver-options "-p 8"` causes Gecode to parallelise to 8 cores.

### 2.5.10 Solver Control – Minion

The following flag specifies where the Minion executable is. The default value is `minion`.

```
-minion-bin <filename>
```

When Minion is run directly by SAVILE ROW, preprocessing is usually applied before search starts. The following flag allows the level of preprocessing to be specified.

```
-preprocess LEVEL
    where LEVEL is one of None, GAC, SAC, SAC_limit, SSAC, SSAC_limit,
    SACBounds, SACBounds_limit, SSACBounds, SSACBounds_limit
```

The default level of preprocessing is `SACBounds_limit`.

The `-preprocess` flag affects the behaviour of Minion in two cases: first when Minion is called to filter domains (the `-reduce-domains` option), and second when Minion is called to search for a solution.

### 2.5.11 Solver Control – Gecode

The following flag specifies where the Gecode executable is. The default value is `fzn-gecode`.

```
-gecode-bin <filename>
```

### 2.5.12 Solver Control – Chuffed

The following flag specifies where the Chuffed executable is. The default value is `fzn-chuffed`.

```
-chuffed-bin <filename>
```

### 2.5.13 Solver Control – SAT Solvers

SAVILE ROW is able to run and parse the output of MiniSAT and Lingeling solvers. The following flag specifies which family of solvers is used. Glucose is the default.

```
-sat-family [minisat | lingeling | glucose]
```

The following flag specifies where the SAT solver executable is. The default value is `minisat`, `lingeling`, or `glucose`, depending on the SAT family.

```
-satsolver-bin <filename>
```

### 2.5.14 Parsing Solutions

Finally, when using ReadSolution mode, SAVILE ROW will read a solution in Minion format and translate it back to ESSENCE PRIME. This allows the user to run Minion separately from SAVILE ROW but still retrieve the solution in ESSENCE PRIME format. When running Minion, the `-solsout` flag should be used to retrieve the solution(s) in a table format.

When using ReadSolution mode, the name of the aux file previously generated by SAVILE ROW needs to be specified using the `-out-aux` flag, so that SAVILE ROW can read its symbol table. Also the name of the solution table file from Minion is specified using `-minion-sol-file`. `-out-solution` is required to specify where to write the solutions. `-all-solutions` and `-num-solutions <n>` are optional in ReadSolution mode. Below is a typical command.

```
savilerow -mode ReadSolution -out-aux <filename> -minion-sol-file <filename>
        -out-solution <filename>
```

When neither `-all-solutions` nor `-num-solutions <n>` is given, SAVILE ROW parses the last solution in the Minion solutions file. For optimisation problems, the last solution will be the optimal or closest to optimal.

## Appendices

### A SAT Encoding

We have used standard encodings from the literature such as the order encoding for sums [8] and support encoding [6] for binary constraints. Also we do not attempt to encode all constraints in the language: several constraint types are decomposed before encoding to SAT.

#### A.1 Decomposition

The first step is decomposition of the constraints AllDifferent, GCC, Atmost and Atleast. All are decomposed into sums of equalities and we have one sum for each relevant domain value. For example to decompose AllDifferent( $[x, y, z]$ ): for each relevant domain value  $a$  we have  $(x = a) + (y = a) + (z = a) \leq 1$  (or  $(x = a) + (y = a) + (z = a) = 1$  when the number of variables and values are equal). These decompositions are done after AC-CSE if AC-CSE is enabled (because the large number of sums generated hinders the AC-CSE algorithm) and before Identical and Active CSE.

The second step is decomposition of lexicographic ordering constraints. We use the decomposition of Frisch et al [5] (Sec.4) with implication rewritten as disjunction, thus the conjunctions of equalities in Frisch et al become disjunctions of disequalities. This decomposition is also done after AC-CSE and before Identical and Active CSE. However, if AC-CSE is switched on, we (independently) apply AC-CSE to the decomposition, thus extracting common sets of disequalities from the disjunctions.

The third step occurs after all flattening is completed. The constraints min, max, and element are decomposed. For  $\min(V) = z$  we have  $V[1] = z \vee V[2] = z \dots$  and  $z \leq V[1] \wedge z \leq V[2] \dots$ . Max is similar to min with  $\leq$  replaced by  $\geq$ . The constraint  $\text{element}(V, x) = z$  becomes  $\forall i : (x \neq i \vee V[i] = z)$ .

#### A.2 Encoding of CSP variables

The encoding of a CSP variable provides SAT literals for facts about the variable:  $[x = a]$ ,  $[x \neq a]$ ,  $[x \leq a]$  and  $[x > a]$  for a CSP variable  $x$  and value  $a$ . We will refer to  $[x = a]$  and  $[x \neq a]$  as direct literals, and  $[x \leq a]$  and  $[x > a]$  as order literals. If the variable has only two values, it is represented with a single SAT variable. All the above facts (for both domain values) map to the SAT variable, its negation, *true* or *false*.

When the CSP variable has more than two values, an encoding is chosen to provide the literals that are needed to encode the constraints. There are three encodings, with one providing the order literals, another providing the direct literals, and the third providing both. Encodings of the binary constraints  $<$  and  $\leq$  use the order literals, and the tree encoding (see Section 2.5.7) of sums also uses the order literals. All other constraint encodings use the direct literals. If the objective function is a single variable, the order literals are required for that variable. Otherwise, if the objective function is a sum at encoding time (only possible with the MaxSAT backend) then direct literals are required for all variables in the sum.

If the order literals are not needed for a variable  $x$ , then the variable is encoded with one SAT variable per domain value (representing  $[x = a]$ ). The 2-product encoding is used for at-most-one and a single clause for at-least-one.

If the direct literals are not needed, then using the language of the *ladder* encoding of Gent et al [7], we have only the ladder variables and the clauses in Gent et al formula (2). If both order and direct literals are required, then we use the full ladder encoding with the clauses in formulas (1), (2) and (3) of Gent et al. Also, for the maximum value  $\max(D(x))$  the facts  $[x \neq \max(D(x))]$  and  $[x < \max(D(x))]$  are equivalent so one SAT variable is saved. For the minimum value  $\min(D(x))$ , the facts  $[x = \min(D(x))]$  and  $[x \leq \min(D(x))]$  are equivalent so one SAT variable is saved. Also, a variable may have gaps in its domain. Suppose variable  $x$  has domain  $D(x) = \{1 \dots 3, 8 \dots 10\}$ . SAT variables are created only for the 6 values in the domain. Facts containing values  $\{4 \dots 7\}$  are mapped appropriately (for example  $[x \leq 5]$  is mapped to  $[x \leq 3]$ ). The encoding has  $2|D(x)| - 3$  SAT variables.

### A.3 Encoding of Constraints

Now we turn to encoding of constraints. Some simple expressions such as  $x = a$ ,  $x \leq a$  and  $\neg x$  (for CSP variable  $x$  and value  $a$ ) are represented with a single SAT literal. We have introduced a new expression type named SATLiteral. Each expression that can be represented as a single literal is replaced with a SATLiteral in a final rewriting pass before encoding constraints. SATLiterals behave like boolean variables hence they can be transparently included in any constraint expression that takes a boolean subexpression.

At-most-one and exactly-one constraints (i.e. a sum of boolean expressions that is  $\leq 1$  or  $= 1$ ) are by default encoded with the 2-product encoding, which is small and tends to perform better than alternatives. Other encodings can be selected by command-line flags (Section 2.5.7).

There is a choice of 5 encodings of pseudo-Boolean and other sum constraints, as described in Section 2.5.7. The MDD, GPW, SWC, and GGT encodings are implemented exactly as described in Bofill et al [4]. Sum-equal constraints are split into sum- $\leq$  and sum- $\geq$  before encoding. All four of these encodings work with automatic AMO and EO detection as described in the CP 2019 paper [2]. The flag `-amo-detect` is used to switch on the AMO and EO detection.

The Tree encoding for sums decomposes each sum constraint using a binary tree, similar to the generalized totalizer. Auxiliary integer variables are introduced for the internal nodes of the tree. A ternary sum constraint is used to connect each internal node to its two children. The ternary sums are then encoded using the order encoding [8]. The Tree encoding does not work with AMO and EO detection. Tree can be superior to MDD, GPW, SWC, and GGT in the case where the original constraint is a sum-equality, because Tree does not first break down the sum-equality constraint into sum- $\leq$  and sum- $\geq$ .

For other constraints we used the standard support encoding wherever possible [6]. Binary constraints such as  $|x| = y$  use the support encoding, and ternary functional constraints  $x \diamond y = z$  (eg  $x \times y = z$ ) use the support encoding when  $z$  is a constant. Otherwise,  $x \diamond y = z$  are encoded as a set of ternary SAT clauses:  $\forall i \in D(x), \forall j \in D(y) : (x \neq i \vee y \neq j \vee z = i \diamond j)$ . When  $i \diamond j$  is not in the domain of  $z$ , the literal  $z = i \diamond j$  will be false. Logical connectives such as  $\wedge, \vee, \leftrightarrow$  have custom encodings and table constraints use Bacchus' encoding [3] (Sec.2.1).

## References

- [1] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, , and András Z. Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 3–12, 2018.
- [2] Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z. Salamon, Josep Suy, and Mateu Villaret. Automatic detection of at-most-one and exactly-one relations for improved sat encodings of pseudo-boolean constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 20–36, 2019.
- [3] Fahiem Bacchus. Gac via unit propagation. In *Principles and Practice of Constraint Programming—CP 2007*, pages 133–147. Springer, 2007.
- [4] Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Sat encodings of pseudo-boolean constraints with at-most-one relations. In *Proceedings of the 16th CPAIOR 2019: Thessaloniki, Greece*, pages 112–128, 2019.
- [5] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 93–108, 2002.
- [6] Ian P. Gent. Arc consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 121–125, 2002.
- [7] Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (ModRef 2004)*, pages 95–110, 2004.

- [8] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.