UPPSALA
UNIVERSITET

UPTEC IT 21022
Examensarbete 30 hp
Augusti 2021

A Linter for Static Analysis of MiniZinc Models

Erik Rimskog

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Linter for Static Analysis of MiniZinc Models

Erik Rimskog

MiniZinc is a modelling language for constraint satisfaction and optimisation problems. It can be used to solve difficult problems by declaratively modelling them and giving them to a generic solver. A linter, a tool for static analysis, is implemented for MiniZinc to provide analysis for improving models. Suggesting rewrites that will speed up solving, removing unnecessary constructs, and pointing out potential problems are examples of the analysis this tool provides. A method for finding points of interest in abstract syntax trees (parsed models) is designed and implemented. The linter is tested and evaluated against models in the MiniZinc Benchmarks, a collection of models used to benchmark solvers. The result from running the linter on one of the models from the benchmarks is more closely inspected and evaluated. The suggestions were correct and made the model simpler, but, unfortunately, there was no noticeable impact on the solving speed.

Handledare: Pierre Flener
Ämnesgranskare: Justin Pearson
Examinator: Lars-Åke Nordén
UPTEC IT 21022
Tryckt av: Reprocentralen ITC

Sammanfattning

Statisk kodanalys är en metod för att fånga många olika sorters fel och buggar innan ett program körs. En *linter* är ett samlingsnamn på verktyg som utför statisk analys, och denna rapport beskriver hur en linter för modelleringsspråket MiniZinc skapades.

Det finns många kombinatoriska optimeringsproblem som är väldigt svåra att lösa då det inte verkar finnas några algoritmer som kan lösa dem på polynomisk tid. Med andra ord finns det inga kända algoritmer som kan lösa dem problemen "snabbt", utan en approximering eller fullständig sökning måste genomföras. Oftast är en fullständig sökning för långsam, det kan t.o.m. i vissa fall ta flera miljoner år att gå igenom alla potentiella lösningar, även för relativt små problem. Två exempel på typiskt svåra problem är schemaläggning och att hitta en ordning att besöka några städer i som ger den kortaste resvägen.

Det finns generella program som kallas för *lösare* som löser ett problem med hjälp utav specialiserade algoritmer och metoder som kan hitta en lösning mycket snabbare än att naivt söka igenom alla lösningar. Lösarna tar oftast en deklarativ beskrivning av problemet de ska lösa, kallad *modell*. Olika lösare är bra på olika sorters problem, så flera lösare behöver testas för ett visst problem för att se vilken som är bäst.

MiniZinc är ett verktyg för att modellera kombinatoriska optimeringsproblem på ett universalt vis: en och samma modell kan användas på flera olika lösare utan att behöva skriva om modellen för varje (i lösarens egna språk). I MiniZinc kan man modellera problem med hjälp utav beslutsvariabler (*decision variables*) och begränsningar (*constraints*) på dessa variabler. En lösares uppgift är då att hitta värden på dessa variabler så att alla begränsingar uppfylls.

Det är lätt att skriva modeller som inte beskriver ett problem i fråga på bästa möjliga sätt. Konsekvensen blir oftast att det tar längre tid för lösaren att hitta en lösning. Lintern beskriven i denna rapport letar främst efter vanliga fel som allmänt är kända att göra det svårare för en lösare att hitta en lösning. Potentiella fel hittas med hjälp utav en egengjord sökare av abstrakta syntaxträd, vilket är ett formellt sätt att representera källkod på.

Lintern evaluerades i detalj på en utvald modell från *MiniZinc Benchmarks*, en samling av modeller för prestandamätning. Modellen blev enklare och mer tydlig, med inga märkbara skillnader på lösningstiden.

Contents

1 Introduction

Static analysis on source code is a way to catch many kinds of errors before a program even is executed. Bugs, weird special cases, and inefficient code are some of many things that are possible to statically check, and the tools that do this are called *linters*. The history of linters is long. The first linter was lint [Joh78], a tool for statically checking the code of C [KR78] programs.

MiniZinc is a declarative solver-independent modelling language for constraint satisfaction and optimisation problems [NSB+07]. The MiniZinc compiler reports syntax errors and other errors that inhibit a successful compilation. It does not suggest improvements on the model itself by, e.g., reporting constructs (syntactically valid sequences of tokens, e.g., `if`-expressions) that should be avoided if possible, or constructs that are error-prone. This is not surprising, since the MiniZinc compiler's main job is to process MiniZinc code into FlatZinc code. If the construct is valid, then the compiler will process it, else abort. The goal of this project is to implement a linter that does these model-improving checks for MiniZinc models. The checks (or rules) implemented are all described in Section 4.

Constraint satisfaction and optimisation problems have many important applications, such as scheduling and finding optimal routes. Problems like these can be difficult to solve as they often do not have a known polynomial-time algorithm. There are programs called *solvers* that are made to find solutions to problems like this using advanced algorithms in different technologies, each one good in its own way. Problems are usually described declaratively as models, where each solver has its own way of specifying them. MiniZinc is an effort to create a modelling language that can be used with many solvers, making it a lot easier to try different solvers with the same problem description. The core of constraint problems consists of constraints and decision variables. A *decision variable* is a variable with unknown value, and a solver's task is to find appropriate values for all of these. Each decision variable has a domain, which is a set of values the variable is allowed to take. The domains can be specified explicitly or, in some cases, be inferred by the MiniZinc compiler. A *constraint* specifies how decision variables relate to each other, or what values are allowed, e.g., that one variable must be greater than another.

Declarative models specify *what* problem to solve, not the exact steps to *how* it should be solved. It is therefore easy to write inefficient models without knowing it, since a small edit can greatly affect solving time. A tool to point out potential issues is attractive to have, especially for beginners, as they might not know what implications each aspect of a model has and its impact on solving time. For example, the domain of an integer decision variable should be as small as possible to limit the search space, but if a domain is not explicitly specified by the modeller, then the domain can potentially be as big as

the solver can handle, which can slow down solving.

To save time, simplify the implementation, and allow for a possible later integration with the MiniZinc project, the parser from the MiniZinc project was reused for this project, specifically its output, namely an abstract syntax tree (AST). It is generally also good practice to reuse components, there is no need to reinvent the same component again. An *AST* is a data structure encoding the formal structure of a program, a MiniZinc model in this case. The MiniZinc compiler is written in the programming language C++ [ISO20], so this project is also written in C++ to be able to use the same parser as easily as possible. Searching for relevant constructs in the MiniZinc AST is one of the main tasks of this project. More details on how the AST is searched are described in Section 3. More detailed explanations on MiniZinc, linters, and parsers are presented in Section 2.

The linter was run on all models in the MiniZinc Benchmarks [Min21b], a big repository of models where most come from earlier MiniZinc Challenges [SFS+14], a competition for solvers. The results are discussed in Section 6. An in-depth look into one of the models in these benchmarks is explored in Section 5, the suggestions from the linter are performed and benchmarked.

Possible future work is presented in Section 7.

2 Background

Section 2.1 give necessary background information about MiniZinc, what kinds of problems it can solve, and how a MiniZinc model is written. What linters are is presented in Section 2.2, and in Section 2.3 are parsers and abstract syntax trees explained.

2.1 MiniZinc

A *Combinatorial problem* is a problem of discrete states where some of these states are solution states [Sch03]. *Combinatorial satisfaction* tries to find these solution states, while *combinatorial optimisation* tries to find a solution state that minimises or maximises some quantity associated with each state.

MiniZinc offers a language for modelling combinatorial problems, i.e., constraint satisfaction and optimisation problems. These are modelled using decision variables and constraints on those variables [RvW06]. Each constraint limits the allowed values of one or more variables. For example, $x_1 > x_2$, or $x_3 = 3$, etc. A solution state from all current variable assignments is formed when all constraints are satisfied.

How a model is processed and used by MiniZinc to produce a solution is explained in

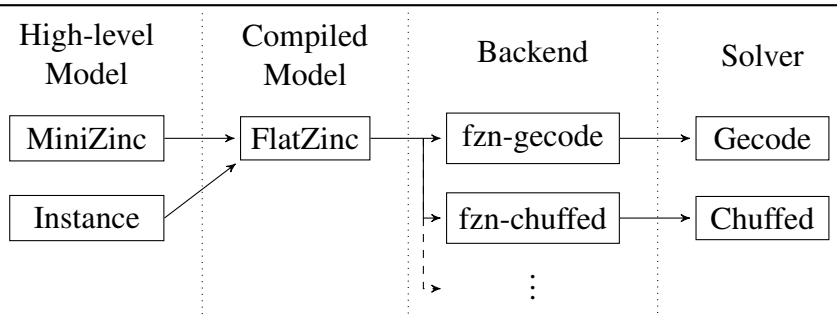

Figure 1 How a MiniZinc model gets run by a solver to produce a solution. The high-level model is compiled (flattened) into a low-level model that a backend and solver pair uses to find solutions.

Section 2.1.1. An introduction to modelling in MiniZinc is presented in Section 2.1.2. A full example of a MiniZinc model that solves a "real" problem is presented in Section 2.1.3.

2.1.1 Solvers

There are many solvers and technologies for solving combinatorial optimisation and satisfaction problems starting from a model thereof, each one good in its own way. Each solver has its own interface, which makes it time-consuming to compare different solvers, as the same problem has to be reformulated for each one. MiniZinc [NSB+07] is a tool chain for creating universal models that can be run on any solver MiniZinc supports, i.e., the *same* model can be used on different solvers without any rewrites.

A diagram showing how a MiniZinc model is processed to eventually run on a solver is shown in Figure 1. The model itself is written in a text file that is compiled (or flattened) into something called FlatZinc, a lower-level representation of the model designed to be easily used by solvers. Each solver has its own backend, which is a program or library that actually converts the FlatZinc model into the solver's own model that then can be used by the solver.

MiniZinc models are often divided into two parts: the model itself and an instance. The model is made generic in the sense that it models some problem, without concrete parameter values. For example, some model could be modelled in terms of a parameter n, but to be solved by a solver must n have a concrete value. Instances, usually in separate files, provide concrete values to these parameters. This distinction makes it easy to run the same model with slightly different values on all parameters. A model is compiled with one instance at a time, so the model has to be recompiled for each instance.

2.1.2 Syntax Overview

A MiniZinc model consists primarily of decision variables, constraints, functions, and `solve`-statements. Decision variables are of various types like `int`, `bool`, and `float`, and the solver should find appropriate values for them. Each decision variable has a *domain*, which is a set of values the variable is allowed to take. Further reduction of allowed values and how the variables should relate to each other are expressed in constraints. For example, the following listing defines a decision variable `x` of type `int`, and a constraint that constrains `x` to have a value strictly greater than five:

```
var int: x;
constraint x > 5;
```

There are two solving modes, specified in a `solve`-statement: satisfaction and optimisation. *Satisfaction* finds values for all decision variables such that all constraints are satisfied, and *optimisation* additionally minimises or maximises some decision variable or expression. For example, adding `solve minimize x` to the listing above would create a model that tries to find the smallest integer value greater than five, which is six. Adding `solve satisfy` would instead find *some* integer greater than five. An optional *search annotation* can be specified on a `solve`-statement, specifying a hint that some solvers take into account. The following is an annotation that hints how `x` should be searched for:

```
:: int_search([x], input_order, indomain_min)
```

The first argument is the array of variables to specify a hint for, the second argument gives the order in which the variables in the array should be searched for, and the last argument specifies how the chosen variables should be constrained given their current computed domains. In this example, `indomain_min`, specifies that the minimum value in the current domain should be tried first.

Variables that are fixed to a given value are in MiniZinc called *parameters*. A parameter is declared with `par`, or no keyword at all, instead of `var`. These parameters are useful since they can be used to generalise a model, and to give descriptive names to constants. For example, the above listing can be rewritten with a parameter as follows:

```
int: n = 5;
var int: x;
constraint x > n;
```

Now the value of `n` can easily be changed to solve a different instance.

This model can be separated into an instance file and a model file by giving the model the declaration `int: n` and the instance file the assignment `n = 5`. Multiple instance files can be created for a single model, which makes it possible to change all parameters

at once.

Arrays allow for a variable amount of variables. The listing below shows an array `xs` consisting of five integer variables. A variable in the array can be accessed separately with a special syntax. For example, the first variable can be accessed with `xs[1]`. To access all variables in an array at once, the built-in function `forall` can be used. The listing below uses this to constrain all variables in the array to be strictly greater than five, it is like writing a universal quantification: $\forall x \in xs : x > 5$.

```
array[1..5] of var int: xs;
constraint forall(i in 1..5)(xs[i] > 5);
```

The `forall` constraint above is syntactic sugar (alternate syntax) for the `forall`-function on a list comprehension. A *list comprehension* is a way to specify a list in a similar manner to set-builder notation: one or more sets or arrays are enumerated, and the elements satisfying an optional Boolean expression are included in the resulting list; the values included are given by an expression on the enumerated elements. An example comprehension that creates a list of squares of even numbers is:

```
[x*x | x in 1..50 where x mod 2 = 0]
```

The `forall` above can thus also be written as:

```
constraint forall([xs[i] > 5 | i in 1..5]);
```

The final feature that will be covered here is functions. Functions allow the modeller to reuse the same code in multiple places. A *function* has a return value, a function body, and zero or more arguments, all of which can have the same possible types as decision variables (`int` etc.). The following listing is a declaration and usage of a function that takes a decision variable as an argument and returns a decision variable:

```
function var int: a_name(var int: x) = <some expression>;
constraint a_name(x) < 69;
```

Some functions called global constraints are special. A *global constraint* is a function defined in a standard library for high-level modelling abstractions. There is a global constraint called `all_different` that constrains all variables in an array to all take different values from each other. Many solvers implement special inference algorithms for global constraints, which means that they can reason a lot better about them. Using a global constraint is preferred over directly using its definition. For example, it is preferred to use `all_different(xs)` instead of `forall(i,j in index_set(xs) where i < j) (xs[i]!=xs[j])`.

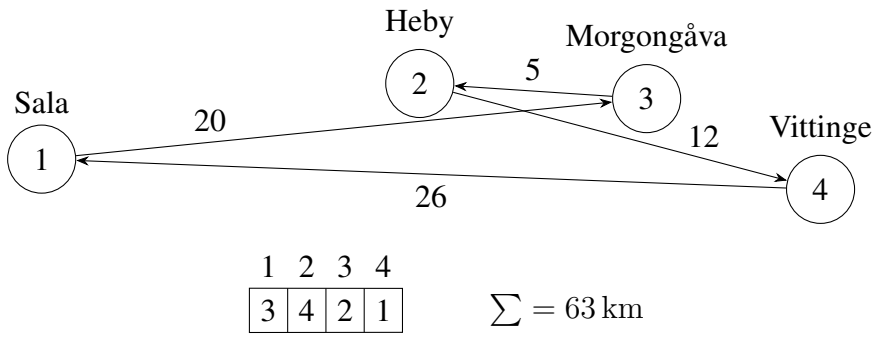

Figure 2 Example instance and non-optimal solution of TSP with $n = 4$ towns. The state is $\langle 3, 4, 2, 1 \rangle$, which means that town 3 is visited after town 1, town 4 is visited after town 2, etc. The total distance travelled is 63 km. Only the paths for the solution route are shown.

2.1.3 Travelling Salesperson Problem

An example of combinatorial optimisation is the travelling salesperson problem (TSP) where a seller wants to travel to $n \in \mathbb{N}^+$ towns in a cyclic order that minimises the total distance travelled. Here, the solution states are all orderings that visit every town once and return to the start town, the quantity to minimise is the total distance.

The following is an example of how TSP can be modelled in MiniZinc, its instance is illustrated in Figure 2:

```
1  include "globals.mzn";
2  int: n = 4;
3  set of int: ns = 1..n;
4  array[ns,ns] of int: dist =
5    [| 0,15,20,26
6     |15, 0, 5,12
7     |20, 5, 0, 5
8     |26,12, 5, 0|];
9  array[ns] of var ns: next;
10 constraint circuit(next);
11 solve minimize sum(i in ns)(dist[i,next[i]]);
```

TSP could be modelled with a list of `n` integers, from 1 to `n`, where each integer represents a town (line 2 and line 3). A 2D-array, or matrix, is constructed on line 4 with the distances between all pairs of towns. The distance between towns `i` and `j` is given by `dist[i,j]`. The actual route to travel is encoded as an array on line 9, that is the decision variables. The value of `next[i]` is what town to visit next; e.g., `next[1] = 2` means that town 2 shall be travelled to after town 1. The global constraint `circuit`

on line 10 is constraining `next` to complete a circuit (cyclic path) between all towns. An example of a circuit is $\langle 2, 3, 4, 1 \rangle$, and an invalid circuit is $\langle 1, 1, 2, 3 \rangle$. The global constraint must be imported from the standard library: see line 1. Finally, line 11 specifies the expression to minimise, which is the total distance in this case. The distance from a town to itself can be anything as `circuit` prohibits a town to have itself as next destination.

2.2 Linters

A *linter* is a program for doing static analysis of source code, *static* meaning that the analysis is done without running the program. The term linter originates from a program called lint [Joh78], a tool for analysing C [KR78] programs. A linter can look for bugs, enforce a certain code style, find a type of improvement, find error-prone constructs, point out special cases, etc. Compilers also do static analysis when they are compiling, e.g., syntax analysis (whether a sequence of characters encodes a valid construct) and semantic analysis (type checking, etc.).

A compiler's main job is to convert source code into some other source code, usually machine code, and it is probably aiming at doing so as fast as possible. A compiler generally only does the minimum amount of analysis required to determine if the given source code is legal, i.e., following the language's standard. Even if a program is legal does not mean it is good: it could crash immediately on start up. Linters take a more leisurely view and try to see whether there are any potential problems with the otherwise legal program. A linter could look for uninitialised pointers in C programs that are dereferenced (as that would most likely make the program crash) and warn about them. Finding uses of variables that have not been set to a value is something the original lint does [Joh78]. Another distinction is that linters are not as precise as compilers. Linters provide *suggestions*: it is fine if they are wrong, and the worst thing that can happen is that the user gets annoyed by false positives. A compiler, in contrast, should never be wrong: it should always be certain about its conclusions, as it might not produce correct programs otherwise.

There also are linters and other static analysis tools for interpreted programming languages, i.e., languages whose programs are run "as is" without compiling them first. Python is an example of a dynamically typed (which means that types are determined at runtime) and interpreted language. It is possible to put optional type information into a Python program and have a third-party program like mypy [Myp21] do static type checking; mypy is not a linter as it performs type checking, something a compiler typically does.

The checks a linter does are typically called *rules*, and different linters have different ways of specifying them. The JavaScript linter ESLint [ESL21] allows the users to

create their own rules in separate JavaScript files, alongside the built-in ones. Since JavaScript is interpreted, new user-specified rules can be added relatively easily: they are loaded as any other file. A rule in ESLint consists of a pattern match on the AST and a function that will do additional checks to see whether it is an actual match or not. The rules in this project are structured in the same manner, except that the part that pattern matches is not as powerful. The one in ESLint can find arbitrary sub-trees, while the one in this project can only find paths (sub-trees that are a line) in the AST.

Clippy [Rus21] is a linter for the programming language Rust [Moz21]. Rust is a compiled language and rules to Clippy are specified as Rust source code. This means that the whole Clippy project has to be recompiled whenever a new rule is added, similar to this project. Clippy's rules are specified similarly to ESLint, with the difference on how the AST is traversed. Each Clippy rule has special functions that are called on various items in the language, like function and `struct` declarations. Each of these special functions determines if there is anything relevant to report to the user for that item.

The linter hlint [hli21] is for the compiled functional language Haskell [Has21]. It specifies rules in YAML, a typical format for general configuration, not in Haskell or a custom-made language. This is possible since the rules for hlint are relatively simple: they are pattern matches against the Haskell source code. The suggested rewrite is also specified as plain Haskell code. One example rule finds `not (a == b)` and suggests rewriting it to `a /= b`, as the rewrite is shorter and handles `NaN` (short for "not a number", typically used for representing invalid results) correctly.

Rules in linters are usually divided into several categories so they can be collectively disabled or enabled more easily. An example of this could be to disable all rules about white space and indentation. They also usually have unique identification numbers or names so that they can be individually disabled if desired. The rules in this project have unique IDs and each rule belongs to some category. The available categories in this project are:

Challenge rules that enforce the rules of the MiniZinc Challenge
Style rules that suggest stylistic changes
Unsure rules that eagerly mark things that in general are bad
Performance rules that suggest changes that can increase performance
Redundant rules that suggest removing redundant or unused things

The tool ShellCheck [She21] is a linter for shell scripts like bash [Ram94]. One rule it has checks for incorrect usage of `$@`. The construct `"$@"` expands to all arguments passed to the current script, which is useful if all arguments should be processed to determine the script's behaviour. The quotes are vital, since the expression without them will perform additional word splitting on the arguments, which is not desired in

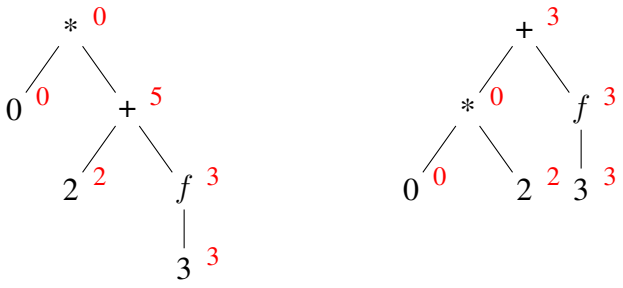

Figure 3 On the left is a possible tree of the expression $0 \cdot (2 + f(3))$. The right one is a possible tree of the expression $(0 \cdot 2) + f(3)$. The asterisk in the trees represents multiplication. The different placements of the parentheses change what the generated tree looks like. The red numbers on the sides represent what each sub-tree evaluates to if $f(x) = x$.

the majority of cases. Consider a script called with the arguments "-f" and "my cool model.mzn". The expression `"$@"` will expand into the same arguments, i.e., "-f" and "my cool model.mzn". However, without quotes, i.e., `$@`, it will instead expand to "-f", "my", "cool", and "model.mzn", because each argument has been split into words. The lint rule called "SC2068" in ShellCheck does exactly this analysis.

2.3 Parsing and Abstract Syntax Trees

To be able to process source code, or models, in text format, it is common to parse them into some kind of data structure. There are many tools available called *parser generators*, which are programs that take some kind of formal description of the language to be parsed and generate source code in some programming language for a parser. MiniZinc uses a parser generator called Bison [Lev09], which is a generator that can generate C++ code, the same language that MiniZinc is written in.

The output of a parser is often an abstract syntax tree (AST), an example of which is given in Figure 3. An AST is representing the formal structure of the source code as nodes with sub-nodes, i.e., a tree. A function in C++ could ,for example, be represented as a node with a child for every statement in that function. Each statement is itself a tree, encoding whatever that statement represents. The information about the arguments and the return type could be stored in the node itself or as yet more children: there are many ways to design an AST.

An AST is abstract in the sense that it does not encode everything in the original code. Things like parentheses in mathematical expressions are often implicitly encoded by the position each operator has in the tree. Two example ASTs of mathematical expressions are shown in Figure 3. Those trees are evaluated bottom up, and each node's value

is its operation applied to the results of its child nodes. Nodes high up in the tree are therefore evaluated *after* their child nodes, which means that sub-expressions that should be evaluated first, like expressions in parentheses, are placed further down in the tree.

3 Implementation

This project is implemented using C++ [ISO20], a compiled programming language with performance as a main design goal. The parser that the MiniZinc project is using was reused in this project, both to save time and to simplify the implementation. Only the abstract syntax tree (AST) will be searched: no extensions or modifications are thus needed, allowing the parser to be reused as is. Using the same AST would also make it possible to reuse other utilities present in the MiniZinc compiler, such as the type checker and pretty printer (converting the AST back to nicely formatted code). Multiple complicated aspects of the language were provided for free as a consequence of reusing the parser, and were thus not needed to be re-implemented. One such aspect is *function resolution*, i.e., the rules and logic behind what function definition to use given a name and some arguments. Another reason for reusing the parser is to make a potential integration into the original project easier, as one parser is easier to maintain than two.

The MiniZinc compiler is written in C++, so creating this project in the same language was a natural choice. C++ comes in several standards, and C++17 is the one chosen for this project as that was the latest one that all major compilers fully supported at the time this project started. The MiniZinc version is 2.5.5 as, again, that was the latest version when this project started.

An overview of how the linter itself is structured is explained in Section 3.1. The AST searcher will be explained in detail in Section 3.2. How rules are added is explained in Section 3.3.

3.1 Application Structure

The overall structure of the linter is illustrated in Figure 4 and everything following here will be illustrated in that figure. The most important component of this linter is a base class called `LintRule`. Each rule is its own class that inherits from `LintRule`. Each rule consists of: a unique ID, what category it belongs to, a name, and a method that accepts a model that returns the result of the rule, if any. This model argument is an object of the class `Model`, i.e., the AST from the parser in the MiniZinc project. The ID and category are used for filtering which rules to use, and the name is used for printing purposes. Each rule performs its analysis by first searching in the AST for a relevant

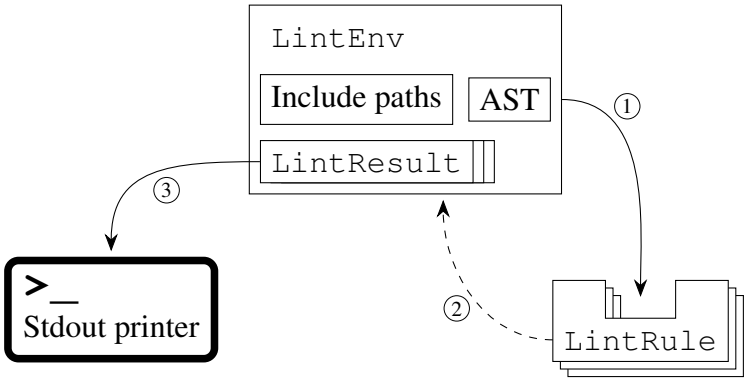

Figure 4 Illustration for the main execution of the linter. The `LintEnv` is given as an argument to each `LintRule` (1). Each rule will analyse the AST and write its results back to a list inside `LintEnv` (2). When all rules have been processed, the list of all results is given to a function that will output them to the user (3).

construct, which, for example, could be a variable used inside the condition of an `if`-expression, followed by multiple checks or other processing to figure out whether this construct should be reported to the user or not. They do this search by using a custom searcher described in Section 3.2 below.

The rules do not take the AST directly: they take it wrapped in a class called `LintEnv`. This class contains a lot of helper functions that each rule interacts with directly. The class `LintEnv` contains, aside from the AST, a list called *include paths*. This list contains file paths to directories that contain library files. MiniZinc comes with a standard library that MiniZinc functions are defined in (e.g., global constraints), and include paths help the linter to find them. The class `LintEnv` also contain all results and methods to create new results. The final thing this class does is caching the results from searches that are common, i.e., multiple rules performs them. One such search is finding all user-defined variables, both top-level ones and those defined in `let`-expressions.

Each result is recorded in a class called `LintResult`. That class hold the relevant data for a result, which is: what rule it came from, the file position (line and column) for where it matched, an explanatory message, and an optional recommended rewrite. These results are then given to a function that will present them to the user. It can be printed to standard output in a human-readable form, or converted to the JSON-format and given to an IDE, or something else.

These are all the relevant components for a high-level overview of the linter. A typical execution of the linter in terms of these components is as follows:

1. Gather options and model files from the arguments of the process.

2. Parse to file into a `Model` class (AST); abort if errors occurred.
3. Type check the AST; abort if errors occurred.
4. Construct a `LintEnv` object.
5. Iterate through all `LintRule` and perform their analysis, saving their results to `LintEnv`.
6. Give the list of all `LintResult` to a function that will output them to, e.g., the terminal.

3.2 Abstract Syntax Tree Searcher

This project uses a custom-made algorithm for finding relevant positions of interest in a MiniZinc abstract syntax tree (AST). Many rules require more complex searches than the matching of single nodes in the AST, e.g., "find `x=3` anywhere inside constraints except for in the `else`-clause of `if`-expressions" instead of "find all `if`-expressions". This algorithm is called *searcher*, and it simplifies the process of searching with complex search queries in the AST. An alternative approach would be to use what the MiniZinc project already uses for processing the AST, namely visitors. A *visitor* would process the tree by running various functions on each node, chosen by the type of the node [GHJV94, p. 331]. It would be easy to find single nodes of interest using visitors, but it would be verbose to find more complicated constructs.

How the ASTs in MiniZinc looks like is explained in Section 3.2.1. How the query language for the searcher works is explained in Section 3.2.2. The algorithm behind the searcher is explained in Section 3.2.3, and its run-time complexity is derived in Section 3.2.4. How more precise searches can be performed is explained in Section 3.2.5. Finally, how the C++ implementation looks like is presented in Section 3.2.6.

3.2.1 MiniZinc Abstract Syntax Tree

The output of the MiniZinc parser is an object of the class `Model`. It is not strictly an abstract syntax *tree*, but more like an abstract syntax *graph*. That is because the `model`-class contains several ASTs that can refer to each other. An illustration of this graph is shown in Figure 5. The `Model`-class contains a list of all top-level items in a model, such as: constraints, variable declarations, and functions. Each of those contains one or more expressions. An *expression* is something that can be evaluated to some value if every variable it contains has a known value. For example, the expression `1+1` evaluates to the integer `2`. There are several *types* that AST nodes in an expression can be of, namely one for each concept, examples being addition, implication, comprehension, and identifier. Each expression is not strictly a tree either since, for example, an `Id`-

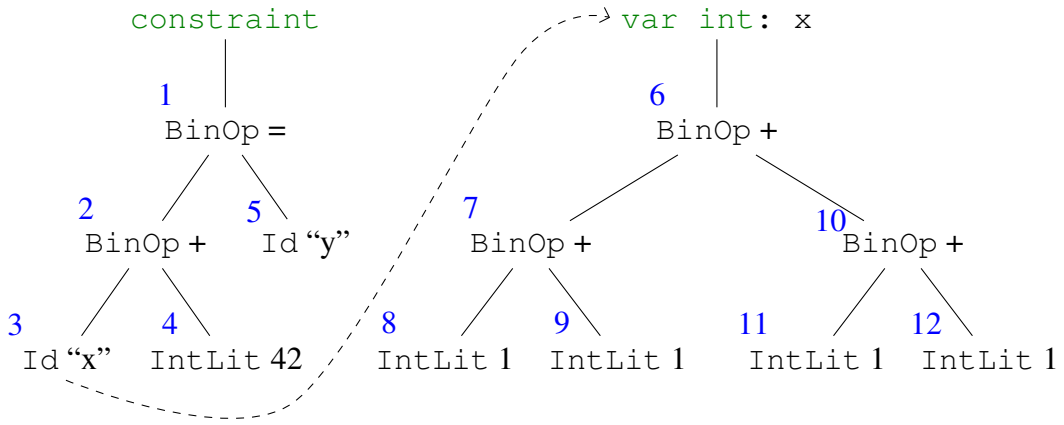

Figure 5 Illustration of two ASTs. The left one corresponds to `constraint x+42=y`, and the right one corresponds to `var int: x=1+1+1+1`. The blue numbers are for referring to the nodes more easily and are not a part of the AST. The `Id` at node 3 has a pointer (the dashed arrow) to its declaration. `BinOp` is short for "binary operation", `Id` is short for "identifier", and lastly `IntLit` is short for "integer literal". These three are names of classes from MiniZinc.

node, which represents an identifier like "x", has a pointer to the variable declaration it is referring to. The searcher explained here will actually search in these top-level ASTs individually, but the whole model will be referred to as one AST for simplicity.

Each expression saves where it came from, namely the filename, beginning line, end line, beginning column, and end column. If an expression does not come directly from a file, but was auto generated by the compiler, then it is marked as "introduced" instead.

3.2.2 Query Language

The searcher finds paths in the AST from a specification, called a *query*. A query consists of two parts: the first one being what top-level items to search in, and the second one being what target path (explained below) to search for in each searched top-level item. Each type of top-level item can either be searched or not, decided by a Boolean value associated with each type.

In a tree, a *path* is a sequence of nodes $n_1, n_2, \ldots, n_m$, where $m \in \mathbb{N}^+$ and n_i is a direct child of n_{i-1} for all $i \in \{2, \ldots, m\}$. A *target path*, given as a non-empty sequence of what are called targets, specifies what a path to search for looks like. A *target* is denoted by $T_{R,N}$, where N is what type of AST node to match against and $R \in \{U, D\}$ specifies how a node matched by a target relates to the node matched by the previous target in

the sequence: the value D (for 'direct') means that the node matched should be a direct child of the previous match, and U (for 'under') means that the node can be a child at any depth of the previous match. If $T_{R,N}$ is the first target in the sequence, then R is not referring to the previous match, but to an implicit dummy node that only has the root as child: the value D in this case means that the match has to be the root of the tree, and U means that the match can occur anywhere. The root refers to the root node of an expression, not to a top-level item since it is not included in the search for paths.

For example, the target path $\langle T_{D,=}\ T_{U,+}\rangle$ will match a `BinOp` = at the root of the AST, and then match a `BinOp` + anywhere on either the left-hand side or right-hand side of the equal-node. Taking the left tree in Figure 5 as an example, this target path will give only the path $\langle 1, 2\rangle$ as a match.

Another example is the target path $\langle T_{U,+}\ T_{U,\texttt{IntLit}}\rangle$: it will match a plus-node anywhere that has an integer literal somewhere under it. The left tree in Figure 5 has one path matching this, namely $\langle 2, 4\rangle$. The right tree on the same figure has several matches, namely $\langle 6, 7, 8\rangle$, $\langle 6, 7, 9\rangle$, $\langle 6, 10, 11\rangle$, $\langle 6, 10, 12\rangle$, $\langle 7, 8\rangle$, $\langle 7, 9\rangle$, $\langle 10, 11\rangle$, and $\langle 10, 12\rangle$.

3.2.3 Algorithm

The algorithm to perform the path matching described in Section 3.2.2 is a backtracking algorithm. It will do a depth-first search (DFS) to iterate over all paths in the AST. If the algorithm at some point deduces that the current path cannot possibly produce a full match for a target path, then it will stop searching that path and backtrack to an earlier point and try a different path.

The pseudo-code is displayed in Algorithm 1 on page 16. The algorithm is given an AST of an expression with $m \in \mathbb{N}^+$ nodes and a target path $t_1, t_2, \ldots, t_n$, where $n \in \mathbb{N}^+$. The algorithm maintains a stack D for a DFS on the AST, and another stack P that contains the current path the DFS is on. Nodes get pushed to P as they are discovered, but the algorithm must also know when to pop them from P. For this, the algorithm has to know when a particular node's children have been searched. Let us say that the DFS popped a new node x from D: it will then push x to P to update the current path, but it will also push it back to D, as a marker, followed by all direct children of x. Now, if at any point later in the search x gets popped again from D and the top element of P also is x, then the DFS algorithm knows that all sub-nodes of x have been processed and that it is now okay to pop P.

Some nodes in P will be matched nodes by some target path t_i, and some will be "nodes on the way" when U is used. Only the matched nodes are of interest, so yet another stack H will keep track of the same path as P but only store nodes matched directly from all t_i. The stack H is therefore a sub-stack of P, and H will have a maximum size of n,

while P will have a maximum size equal to the height of the AST.

A variable $1 \leq x \leq n+1$ is used to keep track of the progress in the target path: the value of x is the index of the next *unmatched* target to find. If $x = n+1$, then all targets are matched. The result in H, when this condition is met, can be saved in a list to later be returned, or it can be processed immediately before the next match is searched for. The latter case was used in this project.

3.2.4 Run-time Analysis

The worst-case run-time complexity of the algorithm presented in Section 3.2.3 mostly depends on how many U there are and what shape the tree has. The worst case is when all targets are U. Consider that the AST is a straight line that is m nodes long and that there are n targets to match. If all t_i targets match on every node, then the searcher will iterate over all $\binom{m}{n}$ matches. An upper bound can be found from the definition:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!} = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \dots \cdot \frac{m-(n-1)}{1} \leq \frac{m^n}{n!}$$

The worst-case complexity is then $\mathcal{O}(m^n/n!)$, at least for when the AST is a line. If the AST was instead a tree, there would not be as many matches since not all $\binom{m}{n}$ choices of nodes are on a path from the root, so it would not be as bad. A discussion on this complexity is presented in Section 6.1.

3.2.5 Filtering

Different kinds of filtering are performed by the searcher. A *filter* is a function $F(n, c)$ that determines whether the child c of a node n should be searched or not. Each target $T_{R,N}$ can have an associated filter that is only executed on nodes the target has matched against: the syntax is provided in Section 3.2.6 below. This can be used to, for example, make sure only the left-hand side of a binary operator is searched upon. In general, it makes a search more specific and fine-tuned. It is also possible to specify *global filters*: those are filters that are executed on every node encountered in a search, including those not directly matched by a target.

Only considering user-defined variables, functions, constraints, etc., is another kind of filtering this searcher does. It is not of interest for the user to get lint results in the standard library, as the user cannot modify it anyway (at least not if the model should be usable by others with an unmodified standard library). Given an include path to the standard library, the searcher will not search in functions and variables whose origin filename is somewhere inside the include path. It will also ignore those whose location is introduced, i.e., they have been generated by the compiler.

```
1:  procedure SEARCHER(AST, t_1, ..., t_n)
2:      D ← root(AST)   P ← ∅   H ← ∅                    ▷ Initialise all the stacks
3:      x ← 1                                             ▷ t_1 is the next target to match
4:      while D ≠ ∅ do
5:          d ← pop(D)                                    ▷ the currently visited node
6:          if d = top(P) then                            ▷ A marker was popped
7:              pop(P)                                    ▷ d is no longer on the current path
8:              if d = top(H) then
9:                  pop(H)
10:                 x ← x − 1
11:                 if t_x is U then      ▷ This step forces d to be unmatched, this allows
                                          ▷ other nodes under d to be matched against t_x
12:                     Push d to P and D, and also push all direct children of d to D
13:                 end if
14:             end if
15:             continue
16:         end if
17:         if t_x matches d then
18:             x ← x + 1
19:             Push d to H
20:         else
21:             if t_x is U then          ▷ t_x had to match directly after t_{x−1}, but it did not
22:                 continue
23:             end if
24:         end if
25:         Push d to P and D             ▷ d is now part of the current path and it is a marker
26:         if x > n then                                 ▷ All t_i have matched
27:             yield H                                   ▷ H contains a whole match, i.e., a result
28:         end if
29:         Push all direct children of d to D
30:     end while
31: end procedure
```

Algorithm 1 The procedure to search for target paths in an AST of an expression. The targets $t_1, \ldots, t_n$ is the target path to search for. The keyword **continue** aborts the current loop iteration early and immediately begins the next one. The keyword **yield** returns a sub-result by temporarily pausing the execution of the procedure.

3.2.6 C++ Interface

An instance of the `Search` class is constructed from a `SearchBuilder`. The `Search` class contains all settings for a search operation, namely: the list of all targets $T_{R,N}$ (see Section 3.2.2), the Booleans for what kinds of top-level items to search in, the global filters, the include paths, and a Boolean for whether all included models should be recursively linted or not. The builder defines many functions for specifying *where* a search should take place and *what* to search for. All functions beginning with `in_` specify what kind of top-level items should be considered: no items are searched in by default. For example, calling `in_constraint` specifies that all constraints should be searched in, and `in_everywhere` enables search in every item. All targets $T_{R,N}$ are added via the functions `under` and `direct`, both of which take the ID of a node to match against. To later retrieve a pointer to a matched node, each target must explicitly be marked to be saved: this is done via `capture`. The final object is constructed via `build` when all customisation is done.

An example that searches for addition nodes (captured) inside equality nodes is constructed as follows:

```
const Search s = SearchBuilder()
  .in_everywhere()
  .direct(BOT_EQ)
  .under(BOT_PLUS)
  .capture()
  .build();
```

The arguments to `direct` and `under` can be anything from the enumerations: `ExpressionId`, `UnOpType` (unary operator type), and `BinOpType` (binary operator type), all of which are defined in the MiniZinc compiler. MiniZinc uses these enumerations to distinguish between the different types of AST nodes.

The variable `s` contains the information necessary for a search, but to actually perform a search is yet another class constructed. Calling `s.search` will return a `ModelSearcher` or `ExpressionSearcher`, depending on if the function was supplied a pointer to a model or a pointer to an expression. Both types of return value implement similar interfaces. The rationale behind this design decision is that `s` now can be used several times without using the builder every time. The difference between both concrete searchers is their granularity: `ModelSearcher` searches on whole top-level items, while `ExpressionSearcher` searches inside an expression at any level in the AST. The latter can be used to do further searches from the results of the former.

The results are iterated one at a time in a lazy fashion by calling `next` on, e.g., `ModelSearcher`. The function returns true if there is a result available. Captured nodes can be retrieved by calling the function `capture`, supplied with a zero-based

index for what node to return. The function `cur_item` returns a pointer to the current top-level item that is currently being searched.

An example where a model is searched is as follows:

```
auto ms = s.search(model);
while(ms.next()) {
  const Expression *plus = ms.capture(0);
  const Item *item = ms.cur_item();
}
```

If no `under` or `direct` is called, then the built searcher is still valid, and it will instead iterate over all top-level items without searching for anything inside expressions. The `SearchBuilder` can also construct `Search` objects that recursively visit included files, which is not done by default. It is also possible to specify a filter for each target, as explained in Section 3.2.5. The filters are added after an `under` or `direct` by calling `filter` with a function pointer as argument (or a lambda without captures).

3.3 Rule Implementation

All rules have their own file in `src/linter/rules/`, and each one defines a class inheriting from `LintRule`. The one for rule "No Domain on Variables" (Section 4.8) is defined in the following way:

```
namespace {
using namespace LZN;

class NoDomainVarDecl : public LintRule {
public:
  constexpr NoDomainVarDecl()
      : LintRule(13, "unbounded-variable",
                 Category::PERFORMANCE) {}

private:
  virtual void do_run(LintEnv &env) const override {...}
};

} // namespace

REGISTER_RULE(NoDomainVarDecl)
```

The class is defined in an anonymous name space, and the constructor for the parent class takes the necessary metadata for the rule, namely its ID, category, and name. The method `do_run` does the actual searching in the AST provided by its argument `env`.

The macro at the very end creates a static instance of the class of which a pointer is given to a static registry. In other words, the rule will be added to a global list of available rules before the main function is executed.

The code for `do_run` in the rule above is currently defined as follows:

```
1  for (const MiniZinc::VarDecl *vd
2       : env.user_defined_variable_declarations()) {
3    if (isNoDomainVar(*vd) && vd->e() == nullptr &&
4        env.get_equal_constrained_rhs(vd) == nullptr)
5    {
6      auto &loc = vd->loc();
7      env.emplace_result(
8        FileContents::Type::OneLineMarked, loc, this,
9        "no explicit domain on variable declaration"
10     );
11   }
12 }
```

All found variable declarations are looped over in the for-loop. Processing all variable declarations is a common operation, so the method on line 2 is a wrapper around a searcher that caches its results. The first function call on line 3 returns true if the variable declaration is an integer or float decision variable without an explicitly specified domain. It is a custom function defined locally in the same file, as follows:

```
bool isNoDomainVar(const MiniZinc::VarDecl &vd) {
  auto &t = vd.type();
  auto domain = vd.ti()->domain();
  return t.isvar()
    && t.st() == MiniZinc::Type::SetType::ST_PLAIN
    && (t.bt() == MiniZinc::Type::BaseType::BT_INT ||
        t.bt() == MiniZinc::Type::BaseType::BT_FLOAT)
    && t.dim() >= 0
    && t.isPresent()
    && domain == nullptr;
}
```

If the variable is equated to another value, then the MiniZinc compiler can deduce the domain from that, in which case it is fine to omit the domain. The cached function on line 4 in `do_run` above searches for constraints that functionally define the variable, e.g., constraints like `constraint x=2`.

If a variable declaration meets all criteria, then the result is added to `env` via `emplace_result`. The first argument specifies how the item at location `loc` should

be printed to the screen. Valid options are: printing nothing, printing several lines without any highlighting, and printing a single line with a portion of it highlighted with, e.g., a squiggly line. In this rule is the single highlighted line option is used. The second argument specifies the location, i.e., filename, start line, end line, start column, and end column. The third argument is a pointer to the rule from which the result originates. Finally, the fourth argument is a `string` with a message explaining what the result is.

An excerpt for a relatively complicated searcher (see Section 3.2) from the rule "Effective 0..1 Variables" (Section 4.4) is given as follows:

```
1  const auto main_searcher = env.userdef_only_builder()
2    .in_everywhere()
3    .under(BOT_IMPL)
4    .capture()
5    .filter([](const auto *impl, const auto *side) -> bool {
6      return impl->template cast<MiniZinc::BinOp>()->lhs()
       ↪ == side;
7    })
8    .direct(BOT_EQ)
9    .capture()
10   .direct(E_INTLIT)
11   .capture()
12   .build();
13
14 const auto off_searcher = env.userdef_only_builder()
15   .direct(BOT_EQ)
16   .capture()
17   .direct(E_INTLIT)
18   .capture()
19   .build();
```

The goal is to find expressions of the form `a=1 -> b=1`. The AST for it can be seen if redundant parentheses are added: `((a)=(1)) -> ((b)=(1))`. The implication is the root node, the left and right child are equal-nodes and both equal-nodes have `a`, `b`, or `1` as children. Nodes `a` and `b` can in fact be any kind of expression, as long as the structure of the AST is the same.

The searcher on line 1 above searches for an implication that has an equal-node as its left node, and that equal-node has an integer literal as any direct child. The expression `env.userdef_only_builder()` is a function that returns a `SearchBuilder` with some modified default settings, e.g., recursively linting included non-library files is enabled. When the main searcher has found a candidate, the right-hand side of the

implication is searched with the off-searcher defined on line 14 above. If that searcher also succeeds, then the overall search is complete if both integer literals are equal to `1` and if `a` and `b` have domains of $\{0, 1\}$. The off-searcher can only find one result or none at all, as it only uses `direct`. Annotated code for the combined search process is as follows:

```
1  auto main = main_searcher.search(env.model());
2  while (main.next()) {
3    //helper function to retrieve the other side
4    //of a binary operation
5    auto otherside = other_side(
6      //another helper that retrieves a capture
7      //and casts it to the specified type
8      main.capture_cast<MiniZinc::BinOp>(0),
9      main.capture(1)
10     );
11   auto off = off_searcher.search(otherside);
12   if (!off.next())
13     continue;
14   ⋮
15 }
```

4 Linting Rules

All currently implemented rules are explained, both why they are beneficial and in some cases also how they are implemented. Each one is described in its own section from Section 4.1 to Section 4.14.

The rules point out constructs that should be avoided if possible. Some operations, like dividing decision variables, are known to generally be bad in terms of solving time. Other rules point out expressions that can be rewritten in another way that is better in some way. Producing fewer FlatZinc constraints is one example of an improvement. Fewer constraints is a good indicator for good performance, but it is not a guarantee that fewer constraints will result in better performance. Another similarly good indicator is the number of decision variables.

The rules vary in complexity and confidence: some rules build graphs and provide confident answers, while others simply mark decision variables in places where it is *generally* bad to have them. Some rules are more of a heads-up than something that should be fixed. Most rules are structured similarly: they perform a search for a relevant set of AST nodes, followed by checks on those nodes to determine whether they should be

reported or not. One important detail is that the rules never modify the AST: they are only searching and reading it.

One design decision was to not rely on instances, i.e., the rules should work even if all parameters do not have values. The rationale is that all rules should be valid for all instances of a given model. This limits the ability of some rules, notably rule "Constant Variable" (Section 4.3), as some conclusions it relies upon are impossible or difficult to prove. These limitations are explained in Section 6.4.

Further reading for more in-depth explanations of the implications of each rule is available in the MiniZinc Handbook [Min21a]. Links to sections in that documentation will be provided where relevant. The MiniZinc version and exact location can for the most part be parsed from these links, which is useful in case they change in the future.

4.1 Arrays are Indexed from One

Arrays that are indexed from 1 are sometimes more efficient and often easier to understand. All user-defined arrays are examined and checked whether all their ranges are set literals that start at `1`. An array indexed from 1 and one that is not is shown in the following listing:

```
array[1..4] of var int: good;
array[5..9] of var int: bad;
```

Flattened arrays always start at 1, so each access to an array that does not start at 1 first has to be translated. If the index is a decision variable, i.e., the `i` in `a[i]` is `var`, then an additional decision variable is introduced: it is constrained to a translation of `i` to an index that starts at 1. More details can be found under "Arrays" in "flattening" in the documentation.[1]

4.2 Compactible If-expression

Some `if`-expressions can be rewritten to a more compact form that produce fewer FlatZinc constraints. All `if`-expressions are searched for and considered. If they are of one of the following forms:

```
var bool: b; var int: z; var int: y;
constraint z = if b then y else 0 endif;
constraint z = if b then 0 else y endif;
```

then they can be rewritten, respectively, to more compact versions using implicit Boolean conversions (false converts to 0 and true converts to 1):

[1] https://www.minizinc.org/doc-latest/en/flattening.html#arrays

```
constraint z = b*y;
constraint z = (not b)*y;
```

4.3 Constant Variable

Decision variables, or an array of decision variables, that are all equated to expressions that are constant should not be marked with `var` as their values do not change: they should be marked with `par` instead. It marks the intent of the identifier more clearly, and makes it easier for the compiler. Some examples where the keyword `var` should be omitted or replaced with `par` are:

```
var int: x = 2;
array[int] of var int: a = [1, 2, 3];
```

This rule also checks if a `var` is constrained to be equal to a `par`-value:

```
var int: x;
constraint x = 2;
```

All equalities that have a decision variable on either side are considered. The equality must also be satisfied, i.e., always be true, which is the case in the listing above. An expression is not always satisfied in, e.g., disjunctions (`\/`). The complete logic to determine if an equality always is satisfied is explained in rule "Non-functionally Defined Variables Not in Search Annotation" (Section 4.9).

This rule also checks if all values in an array are constrained to be equal to `par`-values for simple cases, some limitations being explained in Section 6.4. The formulation in the listing below is searched for, i.e., a `forall` that constrains every element to a `par`-value. The index sets of the array and the comprehension have to be exactly the same, and there cannot be a `where`-clause on the list comprehension.

```
array[1..5] of var int: a;
constraint forall(i in 1..5)(a[i] = 1);
```

4.4 Effective 0..1 Variables

Some expressions can be rewritten to better performing expressions if the domains in question happen to be $\{0, 1\}$. For example, if there are two variables declared as `var 0..1: a` and `var 0..1: b`, then the expression `a=1 -> b=1` can be rewritten to the equivalent `a<=b`. In the same way, `a=0 -> b=0` can be rewritten to `a>=b`.

The produced FlatZinc code of the rewrites is shorter and simpler compared to the originals. The original expressions have one implication each, where each side of it gets reified. This introduces two additional Boolean decision variables, one for each side of

the implication. Additionally, there are three constraints produced: one for each side and a third that constrains at least one of the previous to be true. These constraints encode the expression `a!=1 \/ b=1`, which is logically equivalent to `a=1 -> b=1`. A cleaned-up variant (no annotations, shorter variable names, and no predicate declarations) of the FlatZinc code looks as follows:

```
var 0..1: a;
var 0..1: b;
var bool: X_0;
var bool: X_1;
constraint int_ne_reif(a,1,X_0); % a!=1 <-> X_0
constraint int_eq_reif(b,1,X_1); % b=1 <-> X_1
constraint array_bool_or([X_0,X_1],true); % X_0 \/ X_1
```

The rewrite, on the other hand, does not introduce any additional decision variables and only one constraint for the inequality is produced. The constraint encodes the logically equivalent expression `1*a + (-1)*b <= 0`. The FlatZinc code looks as follows:

```
array [1..2] of int: X_0 = [1,-1];
var 0..1: a;
var 0..1: b;
constraint int_lin_le(X_0,[a,b],0); % 1*a + -1*b <= 0
```

The expressions on either side of the implication (on `a` and `b`) can be arbitrarily complex, as long as their domains can be calculated statically. If either depends on the current instance, then a warning is issued by the linter, since if that parameter is changed, then these rewrites might not be valid any more, so the modeller should think twice before rewriting. An example of this is:

```
var int: a; var int: b;
int: money = -1;
int: workDays = 5;
constraint a=(-money) -> b=(workDays div 5);
```

The suggested rewrite from this rule is not correct for all values of `money` and `workDays`.

Additionally, this rule also suggests rewriting `sum(i in S)(a[i] = 1)` to the equivalent `sum(a)`, if `a` is declared as `array[S] of var 0..1: a`. This one is especially good since the first variant is doing implicit conversions between Booleans and integers (`bool2int`) while the rewrite is not.

4.5 Element Predicate

The function `element`, or predicate, that takes an index (i), an array (a), and a value (v) and denotes true if the element at i in the array a is equal to v. The function `element(i, a, v)` is a more verbose way of writing `a[i] = v`, it is even defined as such. The array access syntax should be used instead to make the model easier to read.

4.6 Reified Global Constraint

A global constraint used in a reified context *can* be slow and inefficient, and this rule will mark all such usages. Being reified means that the satisfaction of the global constraint is constrained to be equal to a Boolean variable, as in:

```
var bool: x = all_different(a);
```

Reification happens when a global constraint is used in a context where it is always satisfied. For example, the global constraint

```
constraint b \/ all_different(a);
```

will be reified into something like:

```
var bool: x = all_different(a);
constraint b \/ x;
```

The following global constraints are not reified since all of them are used in contexts that are always satisfied:

```
constraint all_different(a);
constraint all_different(b) /\ all_different(c);
```

The official MiniZinc documentation talks more about this under "Reified and half-reified predicates"[2] and under "Reification".[3]

The linter will consider all functions from the standard library as global constraints, except for those that do not have to be explicitly included, like `forall`.

4.7 Global Variables in Functions

Using a variable from the global scope in a function is usually confusing, as the whole model (or at least everything required to understand the purpose of the global variable in question) has to be read to understand the function in question. It is also not immediately

[2] https://www.minizinc.org/doc-latest/en/fzn-spec.html#reified-and-half-reified-predicates
[3] https://www.minizinc.org/doc-latest/en/flattening.html#reification

clear what variables a function is constraining and accessing. This rule suggests taking all those variables as arguments to the function instead.

For example, the following:

```
var int: g;
function int: f() = g+1;
constraint f() = 2;
```

is suggested to be rewritten to:

```
var int: g;
function int: f(var int: x) = x+1;
constraint f(g) = 2;
```

Only decision variables are considered since the parameters can only be read from. Also, the parameters have to be understood anyway since they usually are part of a model instance.

4.8 No Domain on Variables

Each decision variable has a domain that specifies what values it can take. It is always recommended to specify a tight domain for `int` and `float` decision variables, as not doings so makes the domain *very* large and hence the solving unnecessarily slow. The following shows a bad and a good case:

```
var int: bad;
var 0..5: good;
```

It is fine to not give a tight domain if a variable is equated with an expression, even another decision variable, as the domain can be inferred from the expression. Constraining a decision variable under equality is also fine. The following shows good cases:

```
var 10..20: good;
var int: also_good = good;
var int: also_good2;
constraint also_good2 = good;
```

This rule marks unequated `int` and `float` decision variables that have no tight domain given. More can be read in "Bounds analysis".[4]

[4] https://www.minizinc.org/doc-latest/en/efficient.html#variable-bounds

4.9 Non-functionally Defined Variables Not in Search Annotation

One of the rules of the MiniZinc Challenge [SFS+14] states:

> Each solve item must be annotated with a search strategy, such that fixing all the variables appearing in the search strategy would allow a value propagation solver to check a solution.

A rough approximation of this is to require all non-functionally defined variables to be included in the search annotation, which is not optional. A decision variable is *functionally defined* if it is uniquely defined in terms of other variables. An example of this is a variable constrained under equality to some expression. In the following listing, the variable `a` is functionally defined in terms of `b` and `c`:

```
var int: a; var int: b; var int c;
constraint a = 5*b + c;
solve
  :: int_search([b, c], input_order, indomain)
  satisfy;
```

If `b` and `c` are fixed to some values, then the value of `a` can be obtained. In this case both `b` and `c` must be included in the search annotation, and it is optional for `a` to be included there. An inlined equality constraint is also a way to functionally define variables, i.e., `var int: a=5*b+c`.

This rule marks all non-functionally defined variables that are *not* in the search annotation. This rule is thus primarily intended for the MiniZinc Challenge.

Functionally defined variables are searched for by finding `BinOp`-nodes of type "equality" with an `Id`-node to a decision variable as a direct child (see Section 3.2). Top-level constraints are searched and the equality can be arbitrarily deep in those, as long as the equality is satisfied. The equality is always to be satisfied if it is under conjunction nodes (`/\` and `forall`), under debugging functions (`trace`), or anything else that guarantees that the equality is satisfied. An example where `a` and `c` are both functionally defined is: `constraint a=b+1 /\ trace("I'm here!", c=a*2)`.

Determining whether arrays are functionally defined is approximated in the sense that they are treated as one unit. This means that only one value in the array needs to be functionally defined for the whole array to be considered functionally defined, e.g., `xs[1]=3`. This limitation is present because it is difficult to know whether all values in an array are functionally defined or not (see Section 6.4). Since the variables of arrays usually are constrained together with `forall`, this approximation does not introduce too many false positives.

Another way a decision variable can be functionally defined is via some global constraints. For example, `count(zs, 2, z)` constrains `z` to be equal to the number of times the value `2` occurs in the array `zs`, which means that `z` is functionally defined. The definition of `count` in the MiniZinc standard library is:

```
predicate count(array[int] of var int: xs,
                var int: y, var int: c) =
  c = sum(x in xs)(x = y);
```

The definitions of all functions used in top-level constraints that are always to be satisfied are searched. If an argument of a function definition is functionally defined, then the corresponding argument in the use is also functionally defined. In the listing above, the argument `c` is functionally defined in the definition of `count`, which means that `z` in the constraint `count(zs, 2, z)` also is functionally defined. The definitions are recursively searched for as long as the uses always are to be satisfied (via `/\`, etc.).

There are some limitations unique to these searches:

- The expression `a=b` functionally defines both variables in terms of each other. At least one of those variables needs to be present in the search annotation, but since both are considered functionally defined, none of them are reported by this rule.
- All functions that return variables, such as the variant `count(xs,y)` that returns the count instead of constraining an argument to the count, are currently not checked for. For example, `id(x)=2`, where `id` is the identity function (returning the argument as is), will not consider `x` to be functionally defined, even though it is.
- The arguments to the function usages can only be the names of top-level variables, since those are the only kinds of variables that also can be specified in a search annotation. There is currently one special case of allowed arguments, namely `array1d`, since it is commonly used in the standard library. The function `array1d` convert an arbitrary-dimensional array into a one-dimensional array: the variables in the returned array are thus the same as the ones in the array given as the argument. For example, in the expression `f(xs)` the argument can be deduced to be the top-level decision variable `xs`. In the expression `f(g(xs))` the origin of the variables in the argument of `f` cannot be deduced in general: it can be `xs` or something else. The special case is if `g` is `array1d`, since the return value of that function is the same as its argument: the whole argument `g(xs)` can be deduced to be the top-level variable `xs`.
- Functions that constrain top-level variables via constraints in `let`-expressions are currently not searched for.

4.10 Operators on Expressions With Decision Variables

Some operators like `div` and `pow` can lead to slow solving when performed on expressions containing decision variables.

Pre-computing the possible results and storing them in a table (by a process called *tabling*) is a good work-around, if the table has a manageable size. Tabling is better since the possible calculations are done once before the solving has started, which gives better propagation in some solvers.

It is also sometimes possible to avoid these operators by reformulating the expression. The following listing shows a disjunction involving two decision variables that is avoided be reformulation:

```
var 5..7: x;
var 4..8: y;
constraint x > 5 \/ y > 4; % original
constraint x + y > 5+4;    % reformulation
```

Operators like `\/` and `->` on expressions with decision variables can also be expensive as they introduce more branching and have worse propagation in solvers, at least in Constraint Logic Programming solvers [RvW06, p. 430].

The operators this rule looks for are: `^`, `div`, `mod`, `/`, `xor`, `/`, `->`, `<-`, `<->`, and `not`. This rule eagerly recommends avoiding these operators on variables, which might produce false positives as some cases cannot be reformulated for higher speed.

4.11 Missing Marking of Symmetry Breaking

Some global constraints like `value_precede_chain` are almost exclusively used to break symmetries in models. Breaking symmetries is important for many (but not all, see below) solving technologies since it speeds up solving by reducing the number of possible solutions that have to be explored. Constraints whose purpose is to break symmetries should be marked as such with:

```
constraint symmetry_breaking_constraint(...);
```

Some solving technologies, like local search, are negatively impacted by symmetry breakers, so solvers that use technologies like that ignore all marked symmetry-breaking constraints. This rule eagerly marks global constraints that are normally used for breaking symmetries, namely: `lex2`, `lex_greater`, `lex_greatereq`, `lex_less`, `lex_lesseq`, `strict_lex2`, `seq_precede_chain`, `value_precede`, `value_precede_chain`, `increasing`, and `decreasing`. More general theory

```
int: K;
int: N;
int: M = let {int: J = 5}
         in J+N;
var 0..K: x;
function var int: f() = x+N;
solve minimize f();
```

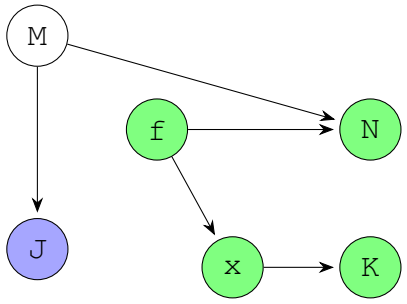

Figure 6 A dependency graph (right) for an example model (left). A directed edge from a node s to a node t means that s is directly depending on t. For example, the variable `M` is using variable `N`. The green nodes (`f`, `N`, `x`, and `K`) are used, and the blue node (`J`) is ignored as it is contained inside the definition of `M`.

about this can be found in "Effective modelling practices".[5]

4.12 Unused Variables and Functions

Variables and function declarations that are not used anywhere should be removed as they are not necessary for the model. Being *used* in this case means to be mentioned inside constraints, the `solve`-statement, the `output`-statement, or other function declarations and variables that are also used.

To calculate whether a variable or function is unused, i.e., not used, a dependency graph is constructed by searching for mentions inside all variables and functions. A *dependency graph* is a directed graph of what variables and functions use what other variables and functions. An example model and dependency graph for it is shown in Figure 6.

After the graph has been constructed, all constraints, the `output`-statement, and the `solve`-statement are checked for any mentions of variables and functions. Each mentioned variable and function, and transitively all their dependants, are marked in green as used. In the example in Figure 6, `f` is used in the `solve`-statement, so `f`, `N`, `x`, and `K` are marked as used. There are no constraints, so `J` and `M` are left as unused.

Reporting both `J` and `M` in this case might seem excessive since `J` is in the definition of `M`. If `M` is unused, then it follows that `J` also is unused. This is even more of a problem with unused functions, as each argument also will be individually reported. To solve this and only report on the outermost variable or function, another graph is constructed: the containment graph. In the *containment graph* are all variable declarations contained in the definitions of other variables, or contained in the bodies of functions, recorded:

[5] https://www.minizinc.org/doc-latest/en/efficient.html#symmetry

each node in the graph represents a function or a variable, and a directed edge from a node a to a node b represents that b is contained inside of a. In this case, only `J` is contained inside another variable, namely `M`. This containment graph is inspected after all variables and functions have been deemed used or unused. Each unused variable or function that is declared inside something else, which also is unused, gets ignored. Therefore, in the example in Figure 6, only `M` is reported to the user as unused.

4.13 Variables in Generators

This rule eagerly reports on all decision variables used in comprehensions, which is generally bad and should be avoided. An example where a decision variable, namely `x`, is used in a generator is:

```
var 1..5: x;
constraint forall(i in 1..x)(...);
```

A `forall` is unrolled into several constraints when flattened, one for each value of `i` in this case. However, if the exact number of constraints is unknown, like here, then the flattener must do more complicated things, and this can slow down the solving. More about unrolling can be read in "Unrolling Expressions"[6] and "Hidden Option Types".[7]

4.14 Variables in If and Where

This rule will eagerly mark all decision variables used in `where`-clauses and in the condition on `if`-expressions, as that can slow down the solving. Examples of this are:

```
var bool: b;
constraint if b then ... else ... endif;
```

```
array[...] of var int: a;
constraint forall(i in ... where a[i] > 5)(...);
```

Having decision variables in these places will make the produced FlatZinc more complex and should thus be avoided if possible. More about the `where` case can be read about under "Hidden Option Types".

[6] https://www.minizinc.org/doc-latest/en/flattening.html#unrolling-expressions
[7] https://www.minizinc.org/doc-latest/en/optiontypes.html#hidden-option-types

nurses_coverage			
Day	Shifts		
1	4	3	1
2	0	0	0
3	0	2	1
4	2	1	1
5	4	4	2
6	1	2	0
7	1	1	0
8	2	2	5
9	4	5	2
10	3	1	3
11	2	0	1
12	2	1	1
13	2	2	1
14	1	2	0

Parameter	Value
n_nurses	16
sched_period	14
n_shifts	3
n_rules	2
period	1..sched_period
shifts	1..n_shifts
shifts_and_off	1..n_shifts+1
nurses	1..n_nurses
rules	1..n_rules
rules_sets	[{4},{3}]
rules_lbs	[1,0]
rules_ubs	[2,1]
rules_windows	[3,3]

Table 1 The values of instance `nsp/period_14/1.dzn`.

5 In-depth Review of a Model

The results from running the linter on a model from the MiniZinc Benchmarks [Min21b] will be explored. The model will be modified upon seeing the linter's suggestions, and each suggestion is benchmarked on some solvers. The model studied is "nurse scheduling problem" (NSP), written by Nina Narodytska in 2007. The unmodified files are provided in Appendix A. The model is described in Section 5.1 and all suggestions from the linter are explored in Section 5.2.

5.1 Model Description

The NSP problem is a satisfaction problem where `n_nurses` are scheduled over a period of `sched_period` days. Each day has `n_shifts` shifts, and each nurse may be assigned to one shift each day. There is also a *coverage* requirement: each shift requires a minimum number of nurses to be assigned to it. Each required coverage is a number between 0 and `n_nurses`. The required coverage is given in the parameter `nurses_coverage`, which is a 2D-array where `nurses_coverage[p,s]` is the number of required nurses for day `p` on shift `s`. The declaration is as follows:

```
array[period, shifts] of 0..n_nurses: nurses_coverage;
```

An example instance is shown in Table 1: it contains the nurse coverage, derived sets, and parameters about regulation rules (explained later). The model has two arrays of

decision variables, shown in the following listing:

```
array[nurses, period] of var shifts_and_off:
↪ nurses_schedule;
array[period, shifts] of var int: coverage;
```

The first array, `nurses_schedule`, specifies for each nurse what shift the nurse should work on for each day. A special value of `n_shifts+1` is used to indicate that the nurse does not work on any actual shift that day. The second array, `coverage`, specifies how many nurses are scheduled to work on each shift each day, and thus has the same structure as the parameter array `nurses_coverage`.

The first constraint in the model specifies that the array of decision variables `coverage` meet the minimum demand in `nurses_coverage`, done as follows:

```
constraint forall (i in period, j in shifts) (
   coverage[i,j] >= nurses_coverage[i,j]
);
```

Next, the two decision-variable arrays are connected: if a nurse `n` on day `p` works on shift `nurses_schedule[n,p]`, then `coverage[p,s]` should reflect that. More precisely, let `s=nurses_schedule[n,p]`, then `coverage[p,s]` should be equal to the number of nurses who are scheduled to work on shift `s` on day `p`. The counts of all occurrences of all types of shifts each nurse works on each day are needed, and for that an array of all types of shifts, used later, which is defined as:

```
array [shifts] of var int: shifts_values;
constraint forall (j in shifts) (
  shifts_values[j] = j
);
```

A predicate `day_distribute` is defined to set the necessary constraints to connect the two arrays for a single day `i`. The predicate is defined as follows:

```
predicate day_distribute(int: i) = let {
  array [shifts] of var int: row_coverage =
    [coverage[i,j]| j in shifts]
  }
  in distribute (
    row_coverage,
    shifts_values,
    [nurses_schedule[j,i]| j in nurses]
    );
```

The array `row_coverage` is the coverage of all shifts for day `i`. The global constraint `distribute` constrains each variable in the first argument to be the number of times

each respective variable in the middle argument occurs in the third argument. For example, `row_coverage[1]` is equal to the number of times `shifts_values[1]` occurs in `[nurses_schedule[j,i] | j in nurses]`, which is an array with the shifts all nurses are scheduled to work on for the day `i`.

Each day is then constrained by this predicate with a `forall`:

```
constraint forall (i in period) (
  day_distribute(i)
);
```

There is one more aspect to this model, as there are `n_rules` regulation rules. Each *requlation rule* specifies the upper and lower bound for how many times a nurse can be scheduled to a set of shifts over a fixed time window. For example, in Table 1 two rules are specified: the first one constrains each nurse to have one or two days off (shift 4) over all sequences of three consecutive days. These rules are added to the model in the same manner as the previous one, i.e., by a predicate and a `forall`. The `forall` is shown in the following listing:

```
constraint forall (i in rules, j in nurses) (
  apply_rule_for_nurse(i, j)
);
```

The following predicate constrains nurse `j` to follow regulation rule `i`:

```
predicate apply_rule_for_nurse (int: i, int: j) =
  let {
    array [period] of var 0..1: rule_for_nurse
  } in
  forall (k in period) (
    (nurses_schedule[j,k] in rules_sets[i])
    <->
    rule_for_nurse[k] = 1
  ) /\
  sliding_sum (
    rules_lbs[i],
    rules_ubs[i],
    rules_windows[i],
    rule_for_nurse
  );
```

This predicate defines an array `rule_for_nurse`: if `rule_for_nurse[k]=1`, then is nurse `j`, on day `k`, scheduled to a relevant shift (`rules_sets[i]`) for rule `i`. The `forall` specifies this relationship for all days.

The global constraint `sliding_sum` constrains all sliding sums of a specified length in an array to be between an upper and lower bound. For example, the following:

```
sliding_sum(0, 420, 2, [1, 2, 3, 4])
```

is the same as the following:

$$\begin{aligned} & 0 \leq 1 + 2 \leq 420 \\ \wedge\ & 0 \leq 2 + 3 \leq 420 \\ \wedge\ & 0 \leq 3 + 4 \leq 420 \end{aligned}$$

5.2 Linter Results

The model was rewritten to satisfy the suggestions from the linter and each rewrite (discussed below) got individually benchmarked. The solvers I had access to were: COIN-BC [FC21] (version 2.10.5/1.17.5), Gecode [Gec21] (version 6.3.0), and Chuffed [Chu11] (version 0.10.4). The solver COIN-BC performed the same, no matter the edit: there were only a few milliseconds of difference between runs, which could be noise. Gecode did not find any solutions under 10 minutes, which is longer than my patience, so nothing can be said about this solver. Chuffed varied drastically in its solve times, which is expected since it is extra sensitive to its input, so no conclusions could be drawn for this solver either. The suggested edits did, however, make the model simpler and hopefully easier to understand.

Each modification to the model will be displayed in a format similar to `diff -u`, i.e., red lines prefixed with − indicate lines from the *old* model that got removed, and green lines prefixed with + indicate *new* lines that got added.

Suggestion 1 The first suggestion is on the constraint with `shifts_values`, where rule "Constant Variable" (Section 4.3) matched. The array `shifts_values` of variables is only constrained to constant values, namely the array `[1,2,3]` for the example instance. There is no need for decision variables, so the array can be rewritten as having parameter in the following way, using a list comprehension:

```
- array [shifts] of var int: shifts_values;
- constraint forall (j in shifts) (
-   shifts_values[j] = j
- );
+ array [shifts] of int: shifts_values =
+   [j | j in shifts];
```

This edit makes it more clear what `shift_values` is, especially since the constraint and variable declaration are separated by multiple lines in the original model file.

Suggestion 2 The second rule to match was "No Domain on Variables" (Section 4.8) on the decision variable `coverage`. This array does not have a tight domain on its variables. Since there is a fixed number of nurses, a shift can never have more nurses assigned to it than the number of nurses. There can also never be a negative amount of nurses assigned to a shift. Hence the following edit:

```
- array[period, shifts] of var int: coverage;
+ array[period, shifts] of var 0..n_nurses: coverage;
```

This variable declaration is hopefully easier to understand now and leads to faster solving.

Suggestion 3 The third rule to match was "Operators on Expressions With Decision Variables" (Section 4.10) on the equivalence in the predicate `apply_rule_for_nurse`. The equivalence, which is a potential source of slowness as its left-hand and right-hand sides often have to be reified, can be removed by utilising implicit `bool2int` conversion on the left-hand side of the equivalence, as follows:

```
predicate apply_rule_for_nurse (int: i, int: j) =
- let {
-   array [period] of var 0..1: rule_for_nurse
- } in
- forall (k in period) (
-   (nurses_schedule[j,k] in rules_sets[i])
-   <->
-   rule_for_nurse[k] = 1
- ) /\
  sliding_sum (
    rules_lbs[i],
    rules_ubs[i],
    rules_windows[i],
-   rule_for_nurse
+   [(nurses_schedule[j,k] in rules_sets[i])
+    | k in period]
  );
```

The model got more compact by this change, so it is hopefully easier to read and faster.

Suggestion 4 Rule "Global Variables in Functions" (Section 4.7) had matches in both predicates, i.e., `apply_rule_for_nurse` and `day_distribute`. They both were using the global arrays without taking them as arguments. Nothing was

changed here as it is unclear whether an improvement in readability would have been made: the predicates are after all used to make their respective `forall` less complicated. There also should not be any performance change at all since all constraints, in the end, are the same.

Additional Edit This is not a suggestion from the linter itself, but instead an observation from applying the previous suggestions. The global constraint `distribute` can take all of its arguments as decision variables. There is a similar global constraint called `global_cardinality` that does the same thing, except that the middle argument can only accept parameters, and the first and third argument swap places. Since `shifts_values` does not consist of decision variables any more can `global_cardinality` be used instead:

```
predicate day_distribute(int: i) = let {
  array [shifts] of var int: row_coverage =
    [coverage[i,j]| j in shifts]
  }
- in distribute (
-   row_coverage,
+ in global_cardinality (
+   [nurses_schedule[j,i]| j in nurses],
    shifts_values,
-   [nurses_schedule[j,i]| j in nurses]
+   row_coverage
    );
```

There is no change in readability (if one is familiar with the unusual distribute constraint), but there could be a performance difference depending on the solver.

6 Discussion

This section will feature discussions on the resulting linter and on my experience using the source code of the MiniZinc toolchain. In Section 6.1 the performance, as in the execution time, of the searcher is discussed. Limitations on the searcher's capabilities and possible solutions to those are discussed in Section 6.2. In Section 6.3 the results of running the linter on all problems in the MiniZinc Benchmarks are explored. Limitations of the static analysis on MiniZinc models are discussed in Section 6.4. Finally, my experience using the MiniZinc parser and source code is presented in Section 6.5.

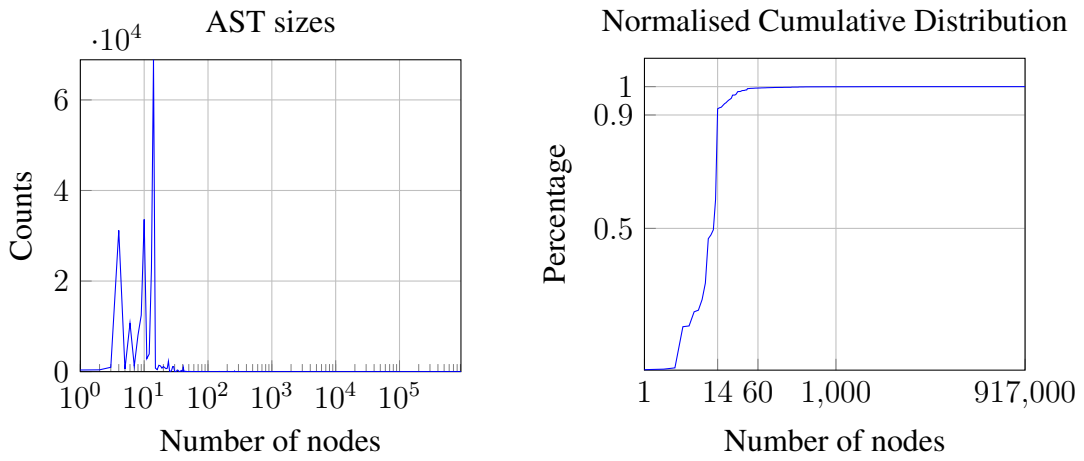

Figure 7 The AST sizes (numbers of nodes they consist of) of all top-level items in all models in the MiniZinc Benchmarks. The left graph shows how many times various AST sizes occur, and the right graph shows a scaled cumulative sum of the left one. The largest AST had a size of 917,442 nodes, a number that occurred 4 times. Most ASTs were small: 92% of all ASTs consisted of 14 nodes or less, and 99.5% consisted of 60 nodes or less. These numbers come from running the AST searcher, configured to match a single node of any type anywhere in the tree, and counting the number of matches.

6.1 Searcher Performance

It took in total around 23 seconds to lint all 299[8] model files in the MiniZinc Benchmarks [Min21b]. It took in total around 13 seconds when only running the parser and type checker. That means that around $(23 - 13)/299 = 0.033$ seconds (33 ms) was spent on linting a model on average. The tests were performed on my average desktop computer. This indicates that the implementation is not too slow as all rules take almost the same amount of time as the parser and type checker combined.

The exponential worst-case complexity presented in Section 3.2.4 does not seem to cause too much of an issue. Most of all top-level expressions in all models in the MiniZinc Benchmarks are relatively small, i.e., they only have a few AST nodes, so the input is not too big to cause an issue. I assume that most other models, aside from these ones, are in the same size range, so the searcher should work well in general. A few top-level expressions had a very large number of nodes though, but they also seemed auto-generated. The sizes of all top-level items are presented in Figure 7.

The exponential behaviour is even less of a problem, since in the final implementation

[8]Was actually 300, but model "mznc2009_roster_model.mzn" contained annotations that do not exist anymore, so it was removed.

only *one* instance of the searcher was using two "under" targets (the cause of the exponential behaviour), as all other ones used one or none. The worst-case analysis also assumes that each target matches all nodes in the AST, but that is not a common occurrence as each target matches a *single* type of node, and there are many different kinds. The searcher will therefore most often than not stop early in the tree, so most of all $\binom{m}{k}$ cases are not even checked.

6.2 Searcher Limitations

The searcher presented in Section 3.2 searches for a path in the AST, and that has been sufficient for most rules, except for the rule "Effective 0..1 Variables" (Section 4.4). That rule has to find expressions of the form `a=1 -> b=1`, but the current searcher can only look for `a=1 -> rhs`, i.e., only one of the sides. This problem was circumvented by starting another search on `rhs` that looked for `b=1`. This particular example is explained more in Section 3.3. Future work could be to implement this kind of functionality on the searcher itself, i.e., adding the ability to search for sub-trees.

Several polynomial-time algorithms for matching (searching) sub-trees are presented in a thesis by Kilpeläinen [Kil92]. The algorithms differ in how the *patterns* looks like, i.e., what is searched for. There is no single algorithm in that thesis that covers all use cases of the AST searcher, but some of these algorithms could selectively be used depending on how the query to the searcher looks like. Simple patterns can use a simpler, and therefore faster, algorithm. The two most interesting algorithms to use in that thesis are "unordered tree inclusion" and "unordered path inclusion". Compared to the current AST searcher, the first matches trees where everything is "under", and the second matches trees where everything is "direct". The first problem is proven in the thesis to be NP-complete, while the second problem has a polynomial-time algorithm. Creating a new searcher that chooses between these two algorithms is not enough since the current AST searcher offers patterns with a mix of both "under" and "direct". The work in that thesis can, however, if anything, be used as a starting point for a new AST searcher that supports searching for sub-trees.

6.3 Reflection on Matches in the MiniZinc Benchmarks

There were many rule hits throughout all models in the MiniZinc Benchmarks, which is not surprising as some rules are general recommendations by design, so most of these hits are probably false positives. It does not mean that they are useless though, as they will make the modeller think twice about, e.g., constraining the division of a decision variable with another. But it might also get annoying to see many yellow squiggly lines in an IDE or text editor. These general suggestions can probably be fine-tuned (to specific solvers) in the future, reducing the number of false positives.

All rules were triggered at least twice when run on all models in the MiniZinc Benchmarks, except for "Variables in Generators" (Section 4.13), which was not hit even once. This indicates that the feature is either: obscure, new, or both.

The rule "Operators on Expressions With Decision Variables" (Section 4.10) and the rule "Reified Global Constraint" (Section 4.6) had many hits, which is expected since those rules are broad and match on expressions eagerly. Doing implications and the like on decision variables is also common, so these rules are maybe too broad.

It was surprising to see that there were models that had no tight domains on their unequated decision variables. There might be domain deductions the MiniZinc compiler can perform that I am not aware of, which these modellers were assuming. There were also some models that equated constant values to some decision variables, which also was surprising.

The final surprising thing was that there were a lot of matches for "Unused Variables and Functions" (Section 4.12): it was in fact the *most* matched one. I did not look at all of them, but there were models that had variables that were only declared and never mentioned again, and those should definitely be removed. Some looked like they were forgotten model parameters from previous iterations of designing the model. I think that this rule will be very useful to have in the future, since it seems to be a common problem. It is a lot easier to understand a model if it only contains what it needs to contain, as there is less to read and less to understand.

6.4 Linter Limitations

Sometimes it is not possible to circumvent a rule without explicitly ignoring it. For example, to produce an array filled with zeros, the comprehension `[0 | i in 1..k]` can be used. But `i` will be reported here as being unused, and there is nothing to do about it. It is a parser error to write `[0 | _ in 1..k]`, which is a common way to ignore variables in other languages. So the lint rule must be ignored here, or a special case should be introduced for that rule.

It is also possible to obfuscate a model to make a rule not match at all. Consider the example in "Effective 0..1 Variables" (Section 4.4), where `a=1 -> b=1` could be rewritten to `a<=b` since both variables have a domain of $\{0, 1\}$. It is equally valid to write `a>=1 -> b=1`, or `(true /\ a=1) -> b=1`, but the linter will not recognise those cases. It is unreasonable to write special cases for all of these since there are so many, and most people will not write `true /\ a=1` anyway.

The independence of specific instances introduces some problems, or rather cases where some conclusion cannot be proved. An example of this is whether a `forall` accesses all values in an array or not. The following model demonstrates a case where it is

impossible to know:

```
int: N; int: K;
array[1..N] of var int: a;
constraint forall(i in 1..K)(a[i] = ...);
```

The `forall` accesses all values if `N` is equal to `K`, but that is not guaranteed to be the case, so the linter will sometimes incorrectly conjecture that the `forall` does not access all values. If the `forall` used the same set as the array (`1..N`), then the linter would know that all values were accessed. Some rules will provide results anyway, even if they depend on parameters, but the linter will warn the user in those cases so an accidental rewrite that breaks the model in the future is not made.

The author of the original lint [Joh78] talks about similar problems with lint, i.e., that some facts are impossible to prove. For example, determining whether a function is unused can depend on runtime data. A function is maybe only executed if a certain file exists, and that is impossible to determine statically, at least in general. So the original lint compromises and only assumes a function is unused if it is never mentioned [Joh78]. This linter makes a similar compromise where the rule "Unused Variables and Functions" (Section 4.12) assumes a function is used if it is mentioned in a constraint. The actual mention could be in a context where it does not matter what the return value of the function is, but the linter will assume that it is used anyway.

6.5 Using the MiniZinc Parser

Not having to create a correct and conforming parser and type checker that followed the MiniZinc language specification made the process of implementing this project easier. There were some quirks that had to be worked around though, as the parser and type checker, ideally, should not be modified. There were, for example, `enum`-declarations from the standard library that looked like they were declared in the main file being linted, and not the file they were originally defined in, so those had to be specially filtered away.

Some generator expressions had the ordering on their `where` clauses swapped around for optimisation purposes *before* flattening. That also made that expression what is called *introduced*, so the original file position information where it came from got lost. That causes problems for the standard output printer, which needs to know where that generator is written so it can print it exactly as is.

The method for figuring out whether something is user-defined or whether it comes from the standard library was not the prettiest: there is not a MiniZinc-provided method for this, so I had to "hack" one using the information available in the AST. As explained in Section 3.2.5, an item is filtered if the file it is defined in comes from an include

path, and comparing file paths is what is not pretty. It is not surprising that a good way of checking that does not exist. The MiniZinc compiler does not really need to know exactly where a function comes from, it only needs its definition.

Integer literals are handled specially in the AST, presumably to save on memory. The literals are stored in nodes of type `IntLit`, but they are *interned*, meaning that nodes with the same contained integer value are shared. For example, all integer literals of value `1` in different locations in the AST refer to the same object in memory. There is a global list that keeps track of all currently allocated `IntLit`. There is also a mechanism that stores the integers *in* the pointers themselves, if they fit, avoiding an indirection. The problem with both of these approaches is that integer literals do not save the location of where they originated from. This means that the standard output printer fails and prints an error if it tries to print a sole integer literal. A way to circumvent this would be to use the pretty printer when printing a location that is flagged as introduced.

A similar annoyance is that the locations of binary operations, e.g., `/\`, are not saved either. The locations of the two arguments are available though, so the location of the operator itself can be approximated by taking the location between the left and the right argument.

7 Future Work

At the moment, the linter executable can only accept a single model-file as an argument, the include path cannot be modified, and the linter will always output the results with colours to the standard output. Some options and configurations should be added to make the linter more useful, notably the ability to choose what rules to use and what to ignore. It is currently possible to disable rules via command-line arguments, but being able to specify that in configuration files as well is also useful. Disabling rules on a line-by-line basis in the model itself via special comments is another useful method to disable rules that should be added in the future as well. There are many more things that are desirable to configure, like the include path, so a proper command-line interface needs to be implemented.

The rules presented in this report are far from everything that is interesting to check for. More rules should be added over time to facilitate future needs, maybe even solver-specific ones for better accuracy. A relatively simple way to create new rules without recompiling the whole program should get some thought. Specifying rules in configuration files similarly to ESLint [ESL21] and hlint [hli21] would be a good way forward. The contribution of new rules to the base linter would also get easier if this more accessible rule creation method were implemented. A flexible and expressive rule specification language would have to be designed to allow as many kinds of rules as possible to be

expressed in it. Extending the AST searcher to allow searching for sub-trees, not only paths, would most likely be needed to achieve a good specification language. Eventually, all rules, even the built-in ones, could be converted to that format.

Many of the ways a model can be obscured in could be counteracted by performing a simplification pass of the AST, before any rules are run. This pass could, for example, find expressions of the form: `false \/ a` and convert them to the equivalent `a`, where `a` is any valid Boolean expression. This pass could also find `[a]` and replace it with `a` if it is type-safe and the semantics is preserved. A specific example is that the rule "Non-functionally Defined Variables Not in Search Annotation" (Section 4.9) cannot deduce that `a` is functionally defined in the expression `[a] = [b+1]`. Solutions could be to introduce a special case in the rule itself, or to run this simplification pass to rewrite the expression as `a = b+1`, which the rule would find. A simplification pass would allow the rules to become simpler, as each one would not need to handle as many special cases.

8 References

[Chu11] G. Chu, "Improving combinatorial optimization," Ph.D. dissertation, Department of Computing and Information Systems, University of Melbourne, Australia, 2011, available at http://hdl.handle.net/11343/36679; the Chuffed solver and MiniZinc backend are available at https://github.com/chuffed/chuffed.

[ESL21] ESLint Team, "ESLint: Find and fix problems in your JavaScript code," 2021, available at https://eslint.org.

[FC21] J. Forrest and Cbc Team, "COIN-OR Branch-and-Cut solver," 2021, official website at https://projects.coin-or.org/Cbc, code at https://github.com/coin-or/Cbc.

[Gec21] Gecode Team, "Gecode: A generic constraint development environment," 2021, the Gecode solver and its MiniZinc backend are available at https://www.gecode.org.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[Has21] Haskell Team, "An advanced, purely functional programming language," 2021, available at https://www.haskell.org.

[hli21] hlint Team, "Haskell source code suggestions," 2021, available at https://github.com/ndmitchell/hlint.

[ISO20] ISO/IEC 14882:2020(E), *Programming Language C++*, 6th ed. Vernier, Geneva, Switzerland: International Organization for Standardization, December 2020, standard available at https://www.iso.org/standard/79358.html, official website at https://isocpp.org.

[Joh78] S. C. Johnson, "Lint, a C program checker," in *COMP. SCI. TECH. REP.* Bell Laboratories, Murray Hill, New Jersey, July 1978.

[Kil92] P. Kilpeläinen, "Tree matching problems with applications to structured text databases," *Department of Computer Science, University of Helsinki*, November 1992, available at https://helda.helsinki.fi/bitstream/handle/10138/21337/treematc.pdf?sequence=1.

[KR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, ser. Prentice-Hall software series. Prentice-Hall, 1978.

[Lev09] J. Levine, *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., August 2009.

[Min21a] MiniZinc Team, "The MiniZinc Handbook," 2021, available at https://www.minizinc.org/doc-latest/index-en.html.

[Min21b] MiniZinc Team and others, "The MiniZinc Benchmarks," 2021, available at https://github.com/MiniZinc/minizinc-benchmarks.

[Moz21] Mozilla Research, "Rust: A language empowering everyone to build reliable and efficient software," 2021, official website at https://www.rust-lang.org.

[Myp21] Mypy Team, "Mypy: Optional static typing for Python 3 and 2 (PEP 484)," 2021, available at http://mypy-lang.org.

[NSB+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *CP 2007*, ser. LNCS, C. Bessière, Ed., vol. 4741. Springer, 2007, pp. 529–543, the MiniZinc toolchain is available at https://www.minizinc.org.

[Ram94] C. Ramey, "Bash, the bourne-again shell," in *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE 1994), The Romanian UNIX User's Group (GURU)*, 1994, pp. 3–5.

[Rus21] Rust Team, "Clippy: A collection of lints to catch common mistakes and improve your Rust code," 2021, available at https://github.com/rust-lang/rust-clippy.

[RvW06] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*. Elsevier, 2006.

[Sch03] A. Schrijver, *Combinatorial optimization: polyhedra and efficiency*. Springer Science & Business Media, 2003, vol. 24.

[SFS+14] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The MiniZinc Challenge 2008–2013," *AI Magazine*, vol. 35, no. 2, pp. 55–60, summer 2014, see https://www.minizinc.org/challenge.html.

[She21] ShellCheck Team, "ShellCheck, a static analysis tool for shell scripts," 2021, source code available at https://github.com/koalaman/shellcheck, official website at https://www.shellcheck.net.

A Nurse Scheduling Model Sources

The raw files of the model described in Section 5 are displayed here. The only modification is to lines 1 and 25 in `nsp_1.mzn`, as those dividers were shortened to fit the page. The directory where all files are located is `minizinc-benchmarks/nsp/` in the Git repository [Min21b] on commit `26bcd0a78433025f7b6896a6fa8caf128795760b`.

A.1 File nsp_1.mzn

```
1  %-----------------------------------------------------
2  % Nurse scheduling problem
3  %
4  % Nina Narodytska
5  % 01.12.2007
6  %
7  % The nurse scheduling problem consists of assigning
   ↪ nurses to shifts to
8  % satisfy  nurses demand for each day.
9  % The model includes the following constraints:
10 % - each shift has a minimum required number of nurses
11 % - various regulation rules on each nurse schedule
12 % To run the model you need a data file from ./period_14
   ↪ or ./period_28 to
13 % specify the minimum number of nurses required on each
   ↪ day[*]
14 %
15 %[*]Original benchmarks come from
   ↪ http://www.projectmanagment.ugent.be/nsp.php
16 % They specify minimal required staff allocation for each
   ↪ shift and individual
17 % preferences for each nurse.  We ignore these preference
   ↪ and replace them with
18 % a set of constraints that model common workload
   ↪ restrictions for all nurses.
19 % The number of nurses in each instance is set to the
   ↪ maximal number of nurses
20 % required for any day over the period multiplied by 1.5.
21
22 % To test the model:
23 %
```

```
% nsp_1.mzn --data test.dzn
%-----------------------------------------------------

include "globals.mzn";
include "test.rules";

%-----------------------------------------------------
% Model parameters
%-----------------------------------------------------


int: n_nurses;          % The number of nurses
int: sched_period;      % The scheduling period
int: n_shifts;          % The number of shifts:
                        %    1 stands for a day shift
                        %    2 stands for an evening shift
                        %    3 stands for a night shift
                        %    4 stands for day-off
int: n_rules;           % The number of regulation rules

set of int: period           = 1..sched_period;
set of int: shifts           = 1..n_shifts;
set of int: shifts_and_off   = 1..n_shifts + 1;
set of int: nurses           = 1..n_nurses;
set of int: rules            = 1..n_rules;


array [period, shifts] of 0..n_nurses: nurses_coverage;
% nurses_coverage[i,j] -- required number of nurses for
  ↪ shift j in day i

% additional regulation rules
array [rules] of set of int: rules_sets;
array [rules] of int: rules_lbs;
array [rules] of int: rules_ubs;
array [rules] of int: rules_windows;


%-----------------------------------------------------
```

```
63 % Model variables
64 %-----------------------------------------------
65
66 array [nurses, period] of var shifts_and_off:
   ↪ nurses_schedule;
67 array [period, shifts] of var int:              coverage;
68 array [shifts] of var int:
   ↪ shifts_values;
69
70 %-----------------------------------------------
71 % Model constraints
72 %-----------------------------------------------
73 %-----------------
74 %-- nurses demand
75 %-----------------
76 %daily nurses demand
77 constraint
78    forall (i in period, j in shifts) (
79              coverage[i,j] >= nurses_coverage[i,j]
80     );
81
82 % encoding for shifts
83 % 1 stands for a day shift
84 % 2 stands for an evening shift
85 % 3 stands for a night shift
86 constraint
87     forall (j in shifts) (
88              shifts_values[j] = j
89     );
90
91 predicate day_distribute (int: i) = (
92    let {
93     array [shifts] of var int: row_coverage =
94       [coverage[i,j]| j in shifts]
95    } in
96     distribute (row_coverage, shifts_values,
       ↪ [nurses_schedule[j,i]| j in nurses])
97   );
98
99 % the distribute constraint
```

```
100 constraint
101   forall (i in period)  (
102     (day_distribute (i))
103   );
104
105 %---------------------
106 %-- regulation rules
107 %---------------------
108
109 % apply a regulation rule to each nurse schedule
110 % ith rule to jth nurse
111 predicate apply_rule_for_nurse (int: i, int: j) =
112    let {
113     array [period] of var 0..1: rule_for_nurse} in (
114     (
115       forall (k in period) (
116         (nurses_schedule[j,k] in rules_sets[i])<->
            ↪ rule_for_nurse[k] = 1
117       )
118       /\
119             sliding_sum (rules_lbs[i], rules_ubs[i],
                ↪ rules_windows[i], rule_for_nurse)
120     )
121 );
122 constraint
123   forall (i in rules, j in nurses)  (
124     apply_rule_for_nurse (i, j)
125   );
126
127
128
129
130 solve satisfy;
131
132
133 %-------------------------------------------------
134 % Output
135 %-------------------------------------------------
136
137
```

```
output [
"Problem description: \n",
"\t number of nurses = ", show(n_nurses), "\n",
"\t scheduling period =  ", show(shifts_values), "\n",
"\t coverage matrix = ", show(nurses_coverage), "\n",
"\t lower bounds = ", show(rules_lbs), "\n",
"\t upper bounds = ", show(rules_ubs), "\n",
"\t windows = ", show(rules_windows), "\n",
"\t sets = ", show(rules_sets),"\n \n",
"Solution: \n",
"\t nurses_schedule =   \n \t", show(nurses_schedule),
↪ "\n"];
```

A.2 File test.rules

```
% Regulation rules

n_rules = 2;
rules_sets = [{4},{3}];
rules_lbs = [1, 0];
rules_ubs = [2, 1];
rules_windows = [3, 3];
```

A.3 File period_14/1.dzn

```
n_nurses =  16;
sched_period =  14;
n_shifts =  3;
nurses_coverage = array2d(period, shifts, [
4, 3, 1,
0, 0, 0,
0, 2, 1,
2, 1, 1,
4, 4, 2,
1, 2, 0,
1, 1, 0,
2, 2, 5,
4, 5, 2,
3, 1, 3,
2, 0, 1,
2, 1, 1,
2, 2, 1,
1, 2, 0]);
```