Automatic Predicate Encapsulation of Potentially Profitably Presolvable Submodels in MiniZinc

Alexander Ek
Abstract

**Automatic Predicate Encapsulation of Potentially Profitably Presolvable Submodels in MiniZinc**

*Alexander Ek*

An underused feature of the MiniZinc language for modelling combinatorial problems is that its models can be parsed, analysed, and modified; however, this could enable modellers to improve their models in novel ways. The auto-tabling tool of Jip J. Dekker allows modellers to annotate a predicate definition in a MiniZinc model for automatic presolving that tabulates the solutions to the predicate within an extensional constraint and replaces the predicate definition with it. This is a well-known and powerful way to reformulate a constraint model, and this auto-tabling tool eliminates a lot of the hassle that comes with doing this reformulation manually. A part of a model that can be presolved is called a submodel, and a submodel can be encapsulated within a predicate definition in MiniZinc.

However, there are still some unanswered, unvisited, but important questions: What are (and how do we find) the submodels that yield the best performance improvement when presolved, and is there a systematic way of identifying, ranking, and encapsulating submodels of a MiniZinc model that can be automated? In this thesis, we present concepts and an implemented tool that answer all the aforesaid questions, and show that the automatically generated results of the tool are similar to those created manually.
Acknowledgements

I would like to thank my supervisor, Justin Pearson, and my reviewer, Pierre Flener, for the help; guidance; encouragement; important comments, feedback, and remarks; and (sometimes a bit off-topic, but nevertheless interesting) discussions during the course of this thesis.
## Contents

1 **Introduction** ........................................... 9  
  1.1 Problem Description ................................... 10  
  1.2 Outline and Contributions .............................. 11  

2 **Background** ............................................ 12  
  2.1 The Constraint Satisfaction and Optimisation Problems 12  
  2.2 The Black Hole Patience Problem ........................ 15  
  2.3 Constraint-Based Modelling ............................. 15  
  2.4 MiniZinc ............................................. 16  
    2.4.1 The MiniZinc Language ......................... 17  
    2.4.2 The MiniZinc Toolchain and FlatZinc ............ 19  
  2.5 Submodel Presolving with Auto-Tabling ................ 20  

3 **A Formalisation of the MiniZinc Language** .......... 24  
  3.1 Syntax ............................................. 24  
  3.2 Semantics .......................................... 29  
  3.3 Base Language: Atomic Functions and Predicates .... 35  

4 **Theory and Design** ................................... 35  
  4.1 Submodel Identification .............................. 35  
    4.1.1 Subformulas .................................. 37  
    4.1.2 Abstractions ................................... 39  
    4.1.3 Generalisations ............................... 41  
    4.1.4 Submodel Identification Graphs ................ 43  
  4.2 Predicate Encapsulation of Submodels ................. 44  
  4.3 Submodel Ranking and Heuristics .................... 46  

5 **Implementation and Application** ..................... 52  
  5.1 Features and Functionality ........................... 53  
  5.2 Overview of the Design ................................ 54  
  5.3 Representation ...................................... 54  
  5.4 Translator: Syntactic and Semantic Analysis ........ 54  
  5.5 Application of Theory: Identifying, Ranking, and Encapsulating 56  
    5.5.1 Submodel Identifier ........................... 56  
    5.5.2 Submodel Ranker ................................ 58  
    5.5.3 Submodel Encapsulator .......................... 58  

6 **Evaluation** .......................................... 59  
  6.1 Setup .............................................. 60  
  6.2 The Block Party Metacube Problem ..................... 60  
  6.3 Runtime Performance of the Encapsulation Process .... 61  
  6.4 Comparison of Manually and Automatically Encapsulated Models 61
7 Discussion

7.1 Discussion of the Research Questions ........................................ 66
7.2 Discussion of our MiniZinc Logic ............................................. 67
7.3 Discussion of the Theory and Implementation ............................... 67
7.4 Discussion of the Evaluation ................................................... 68

8 Related Work

9 Conclusion

10 Future Work

10.1 Future Work on our MiniZinc Logic ........................................ 69
10.2 Future Work on the Theory and Implementation .......................... 69

A Black Hole MiniZinc Model Licence ........................................... 70

B Installation Instructions ............................................................. 71

C A Theory of Constraint Satisfaction Families ................................. 71

C.1 Introduction and Definitions .................................................. 72
C.2 Constraint Graph-like Structures for Constraint Satisfaction Families .......................... 76
C.3 Discussion and Future Work ................................................... 79
1 Introduction

Combinatorial optimisation (and satisfaction) problems are problems where the goal is to find an optimal (or satisfying) arrangement or selection of discrete mathematical elements belonging to finite sets. The Sudoku puzzle is an example of a simple combinatorial satisfaction problem. In the Sudoku puzzle, every satisfying arrangement is the numbers put in each cell such that the rules, which are explained in Section 2.1, are not violated. Combinatorial problems appear in a wide range of areas and have many applications within industry and science; these applications include vehicle routing, air-traffic management and planning, scheduling or sequencing of tasks, rostering and scheduling of staff, timetabling, tournament design, container packing, and medical image analysis. As a result, combinatorial problems are interesting for both academia and industry to study and efficiently solve.

Combinatorial problems are often very computationally challenging to solve since many of these problems are \(NP\)-hard \[1\]. This means that no algorithm can solve these problems in polynomial time over the size of the problem instance, unless \(P = NP\), which many doubt. Hence, at some point, problem instances become too big and practically unsolvable within a reasonable time. It is possible, however, to delay the threshold for where a problem takes too long to solve, but it can be a very extensive and research-heavy task to design such efficient algorithms for solving a specific problem. In response, different types of solving technologies and solvers have emerged to keep people from (re-)designing algorithms based on the same principles and techniques. These solving technologies include SAT and SMT solving \[2\], integer programming \[3\], and constraint programming \[4\].

The nature of solving technologies, and to some extent solvers, is that they delay the threshold differently well for different problems and problem instances. In order to determine the most suitable solver for a problem, empirical research is required since, in general, no solver (or solving technology) dominates all others. This is problematic because a solver usually has a solver-specific modelling language, and hence one often needs to rewrite the model in order to test another solver or solving technology. To address this complication, the MiniZinc toolchain \[5, 6\] has been created to act as a universal modelling language. The goal is to allow the same MiniZinc model to be used by several solvers, which would make it easier to test different solvers and solving technologies.

A MiniZinc model is expressed in terms of decision variables and constraints using high-level predicates (such as all-different and bin-packing). We call this kind of modelling constraint-based modelling and its models constraint models. The decision variables express the unknowns of a problem, e.g., what number to write in an empty cell of a Sudoku puzzle; and the constraints express what is allowed and what is not, e.g., that all numbers must be unique in a row in the Sudoku puzzle.

The nature of combinatorial problems is that the same problem can be expressed by multiple differently formulated constraint models. How a constraint model is formulated can drastically affect how efficiently it is solved by a solver. A well-known and powerful
way to reformulate a constraint model is to precompute the solutions to a part of the model, and tabulate them within an extensional constraint that replaces that part of the model. This means that some parts of a MiniZinc model can be beneficially solved before the rest of the model is. This is called submodel presolving. This technique is tedious to perform and, when used, may obfuscate the original model; for these reasons, it was seldom tried. However, recently an auto-encoding tool for the process has been presented [7], eliminating the tediousness and obfuscation.

1.1 Problem Description

A part of a model that can be presolved is called a submodel, and every submodel can be encapsulated within a predicate definition (for the remainder of this thesis, this will often simply be referred to as encapsulated) in MiniZinc. The auto-encoding tool of Jip J. Dekker [7, 8] allows modellers to annotate predicate definitions in a MiniZinc model for automatic presolving. This eliminates many of the problems regarding submodel presolving, but some are still unanswered: What are the submodels of a model, and how do we find them? What makes a submodel profitably presolvable, and are there some general rules that can be systematised? What are the most profitably presolvable submodels of a model? How can we refactor a MiniZinc model to encapsulate a given submodel?

As will be seen throughout this thesis, for most models there is a large number of submodels and there are also several ways to encapsulate each of them. As a result, it can be a very hard task to determine which submodel will improve the runtime the most (or, at least, by a meaningful amount) if encapsulated and presolved. Knowing the best submodels to presolve in a model is based on three approaches: (1) general intuition, (2) knowledge about the model, and (3) experimentation carried out on the model. However, it is often tedious and practically impossible to perform extensive experiments that include all submodels (or at least all of the very promising ones) of a constraint model since the modeller needs to go through multiple problem instances and submodels. Furthermore, it can also be very tedious to rewrite the model every time the modeller wishes to try and presolve another submodel. And the modeller may not have strong enough intuition or enough knowledge about the model to identify good submodels to try and encapsulate, either. Or, perhaps, the most profitably presolvable submodels may surprise most modellers.

Then, how possible is it to automate these approaches? The last approach is not that hard to automate, but the process would most likely take a long time since multiple problem instances and multiple submodels need to be investigated. It depends, of course, on how fine-grained the conclusion has to be. The second approach is very hard to automate, since it is nearly impossible for an algorithm to infer what problem a model represents and what the most interesting instances of it are; however, it could be possible to let the modeller input extra information about the model during the analysis, or perhaps machine learning could be used here. The first approach seems to be the easiest one to automate, and could be useful when using the third approach as well. The intuition could be distilled into some general rules that in turn can be transformed into
heuristics that score submodels on how profitably presolvable they are predicted to be.

By automatically recommending submodels to try and presolve, the modelling process is sped up, made more available, and may improve models from even the best modellers. And no tools have been made available that can automatically suggest encapsulations.

1.2 Outline and Contributions

An underused feature of the MiniZinc language is that its models can be parsed, analysed, and modified; however, this could enable modellers to improve their models in novel ways. By analysing the syntactic structure and semantics of a MiniZinc model we can create an automatic method for identifying submodels that are profitable presolvable.

In this thesis, the intuition and initial observations of what makes a predicate profitably presolvable are collected and, based on this collection of information, we present methods for automatically encapsulating promising parts of a model into predicates, for presolving. The necessary concepts for the rest of this thesis are presented in Section 2 and the main contributions of this thesis are as follows:

- A formalisation of the MiniZinc language is presented (Section 3). This formalisation is incomplete but is enough to reason on for the purposes of this thesis.
- Using this formalisation, theory for and a systematic approach to identify submodels, rank them, and encapsulate a selected submodel into a predicate definition, are presented (Section 4).
- The implementation of a tool that applies the theory to MiniZinc models is described and explained in Section 5. The tool is shown, in Section 6, to improve models and produce similar encapsulations to those created manually by Dekker et al. [7, 8].

In the last portion of this thesis, we discuss possible future implications, how well this thesis has answered the questions of the problem statement, and the choices made in this thesis (Section 7). This thesis is compared to related work within the field (Section 8), and the facts and results of this thesis are summarised (Section 9). Lastly, open questions are presented along with directions for future work (Section 10).

In Appendix A the licence for the Black Hole MiniZinc Model that is used throughout this thesis is presented. In Appendix B, a link to the source code and installation instructions of the implemented tool can be found. In Appendix C we present a theory that formalises families of instances of constraint satisfaction and optimisation problems. Although it was an alternative approach to identifying submodels to presolve that was abandoned (for reasons explained in that appendix), it is theoretically interesting and may become useful in identifying submodel to presolve if given some more thought.
Figure 2.1: A Sudoku puzzle.

Figure 2.2: A solution to Figure 2.1, where boldface numbers are the clues.

2 Background

In this section, preliminaries relevant for the theoretical framework shown in Sections 3 and 4 are presented. We give an introduction to the constraint satisfaction and optimisation problems (Section 2.1) and an introduction to the black hole patience problem (Section 2.2), which will be a recurring example throughout this thesis. Constraint-based modelling is explained and we compare it with the constraint satisfaction and optimisation problems (Section 2.3). The MiniZinc language and its toolchain are introduced and explained, and how this relates to constraint-based modelling is also explained (Section 2.4). Lastly, an introduction to submodels and submodel presolving is given (Section 2.5).

2.1 The Constraint Satisfaction and Optimisation Problems

The constraint satisfaction problem (CSP) is a particularly interesting combinatorial problem and has been the subject of a lot of research. It is interesting because many instances of other combinatorial problems can be expressed as instances of the CSP. Consider a Sudoku puzzle, such as Figure 2.1, which is a combinatorial satisfaction problem and can be expressed as a CSP instance. **Sudoku** is a number puzzle on a $9 \times 9$ grid of 81 cells, which are divided into nine, non-overlapping, $3 \times 3$ regions of nine cells each. A puzzle is solved when each cell is filled with a natural number of the inclusive interval from 1 to 9, where some of the cells are already filled, called the clues of the puzzle. A puzzle is solved when the following rules are satisfied:

- every row contains exactly one of each number from the interval;
- every column contains exactly one of each number from the interval; and
- every region contains exactly one of each number from the interval.

This formulation of Sudoku is very similar to how a CSP instance is formulated. A CSP instance consists of a set of unknowns (called decision variables) and constraints
on said decision variables where the goal is to assign each decision variable some value such that all the constraints are simultaneously satisfied. In the Sudoku example, the decision variables represent the cells, and the variable domain (i.e., the set of values a decision variable can be assigned) of each decision variable is the numbers between, and including, one and nine. And, finally, the constraints represent the three rules of the Sudoku. Note that variable domains can be replaced with corresponding constraints.

The nature of the CSP allows multiple CSP instances to be interpreted to represent the same concrete problem instance. For example, the rules of the Sudoku can be represented with several different sets of constraints. Furthermore, the decision variables can represent different aspects of the problems and still capture the problem as a whole. Even with the Sudoku, the decision variables need not represent the cells: they could represent the numbers between, and including, one to nine and each be assigned a set of cells. Here, the variable domain of the decision variables is all the subsets, of cardinality nine, of the cells. Note that the constraints need to be formulated in a rather different way here, but, even here, multiple different sets of constraints represent the rules of Sudoku.

The power set (i.e., the set of all subsets) of a set $S$ is denoted $\mathcal{P}(S)$.

**Definition 2.1** (CSP). An instance of the constraint satisfaction problem (CSP) is a 3-tuple $\langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C \rangle$ where:

- $V$ is a finite set of decision variables. Each decision variable $v \in V$ has a variable domain $d(v) \subset \mathbb{Z}$ that is a finite set of possible values that $v$ can be assigned.
- $C$ is a finite set of constraints. Each constraint $C \in C$ is a tuple $\langle S, R \rangle$ where:
  - $S$ is the constraint scope of $C$ and the a set $\{v_1, v_2, \ldots, v_m\}$ of $m$ decision variables that $C$ constrains, where $m$ is called the arity of the constraint $C$.
  - $R$ is the constraint relation of the constraint $C$ and is a finitary relation $R \subseteq d(v_1) \times d(v_2) \times \cdots \times d(v_m)$ over the domains of its constraint scope.

The goal of a CSP instance is to assign each decision variable a value from its domain such that all the constraints are satisfied.

For example, a solution to the Sudoku puzzle shown in Figure 2.1 can be seen in Figure 2.2. Indeed, all the constraints are satisfied and each variable is assigned a number from its domain.

For every function $f : X \rightarrow Y$, given a subset $A \subseteq X$, we denote $f[A] = \{f(x) | x \in A\}$ to be the image of $A$ under $f$.

**Definition 2.2** (Solution). A solution to a CSP instance $\Psi = \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C \rangle$ is a function $s : V \rightarrow d[V]$ that maps every decision variable $v \in V$ to an element from its variable domain $d(v)$ such that all constraints are satisfied, that is, $\langle s(v_1), s(v_2), \ldots, s(v_m) \rangle \in R$ for every constraint $\langle S, R \rangle \in C$ where $S = \{v_1, \ldots, v_m\}$.

1Note that $\mathbb{Z}$ is the domain universe. This is chosen solely for brevity and clarity. In MiniZinc, the domain universe is more complicated; however, the concepts presented in this thesis can still be applied as effectively and efficiently.
If no solution exists, then the CSP instance is unsatisfiable. Consider a Sudoku where the clues violate one of the constraints: such a Sudoku instance is unsatisfiable; however, there may be more deceptive instances where the constraints are not violated by the clues but there is no way of filling in the numbers without violating a constraint.

The search space of a CSP instance is the set of all possible ways to assign each decision variable an element of its domain. The solution space of a CSP instance is the elements of the search space that satisfy all the constraints.

Some combinatorial problems are optimisation problems, not satisfaction problems. Consider the problem where one needs to find, given a map of cities and the roads between them, a route that starts at one city, passes through every other city exactly once, and ends up at the starting city again. This is the Hamiltonian cycle problem. Now, consider the problem where one needs to find a shortest or fastest such route. This is the travelling salesperson problem. Most of the time, the latter problem is more interesting.

The constrained optimisation problem (COP) is also a particularly interesting combinatorial problem since many instances of other combinatorial optimisation problems can be expressed as instances of the COP.

**Definition 2.3 (COP).** An instance of the constrained optimisation problem (COP) is a 4-tuple \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C, f \rangle \) such that \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C \rangle \) is a CSP instance and \( f \) is an objective function. The objective function is defined as a map \( f : d(v_1) \times d(v_2) \times \cdots \times d(v_m) \rightarrow \mathbb{R} \) from domain values of a subset \( \{v_1, \ldots, v_m\} \subseteq V \) of decision variables, called the objective scope, to an objective value. The goal is to find an optimal solution with regard to the objective value (that is, either minimal or maximal objective value, depending on whether it is a minimisation or maximisation problem). Let \( \text{scope}(f) \) denote the objective scope of an objective function \( f \). □

Note that if \( f \) is a constant function, then the COP instance \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C, f \rangle \) is equivalent to the CSP instance \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C \rangle \); hence we call them both a CSP instance. Solutions are defined for COP instances and CSP instances analogously; however different types of solutions need to be distinguished when dealing with COP instances. The goal of COP instances is to find as good solutions as possible, that is, with minimal (or maximal, depending on the goal) objective value.

**Definition 2.4 (Optimal solution).** An optimal solution to a COP instance \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C, f \rangle \) is a solution \( s \) to \( \langle V, d : V \rightarrow \mathcal{P}(\mathbb{Z}), C \rangle \) such that \( f(s(v_1), s(v_2), \ldots, s(v_m)) \), where \( \text{scope}(f) = \{v_1, v_2, \ldots, v_m\} \), is a global optimum of \( f \) across all possible solutions. □

A solution that is not optimal is called a suboptimal solution. Note that if there are multiple optimal solutions, then an optimal solution may be chosen arbitrarily.

Hereafter, we will only consider CSP instances, for brevity and clarity, unless COP instances are important (where it will be explicitly stated) since most of the concepts covered are equivalent for CSP instances and COP instances.
2.2 The Black Hole Patience Problem

Throughout this thesis, the black hole patience problem (BHPP) will be used to explain the presented concepts and definitions.

The BHPP is based on the patience game Black Hole [9], as the name implies, and has been studied in the context of combinatorial optimisation and modelling [10]. The BHPP consists of a full deck of standard playing cards arranged in 17 fans, of three cards each, orbiting around the Ace of Spades placed in the middle (i.e., in the black hole). In the patience game, the player is only allowed to move cards from the tops of the fans to the top of the black hole pile such that the card moved is one rank above or below (regardless of suit) the top-most card already in the black hole pile (ranking is contiguous between Ace and King). If all 52 cards are successfully placed in the black hole, the game is won. A solution to an instance of the BHPP is a sequence of card movements that fulfil the aforesaid rules.

Each instance of the BHPP is a distribution of the playing cards (except the Ace of Spades) between the 17 fans and a permutation of the playing cards within each fan.

2.3 Constraint-Based Modelling

Constraint-based modelling is a declarative style of modelling combinatorial problems, which makes it possible to express a problem essentially as a set of CSP instances or a set of COP instances. The style is inspired by and based on the constraint programming paradigm. The terminology used in constraint-based modelling overlaps with the terminology of the CSP and COP and may sometimes mean different things; however, the concepts are highly similar.

The declarative style means that, like in CSP or COP instances, only the logic of what a solution looks like is expressed, not how it is found. A separate solver, which is the software component that actually solves the model, decides, based on known algorithms and perhaps heuristics, how the model will be solved. It is often possible to express the same constraint in multiple ways, and hence affect how the model is solved. Some solvers allow the modeller to some degree influence how a model is solved. Below, the different concepts used in constraint-based modelling are introduced.

A problem is the description or specification (often in natural language) of what is to be solved and how a solution looks like. For example, CSP, Sudoku, and BHPP are all problems in this sense. A problem may be expressed as a function of parameters. Each different value of these parameters would create a different instance of the problem. For example, knowing where and what the clues are of a Sudoku puzzle, or if there are any clues at all, is necessary for solving that specific Sudoku puzzle. In this thesis we only focus on combinatorial problems, that is, problems with a discrete search space.

A model is an encoding of a problem (in a specific modelling language) and many models may express the same problem. In this thesis, the focus is on constraint models, which are models that are expressed in terms of decision variables and constraints, and potentially with respect to one or more parameter (these concepts are explained in more detail below). Constraint models closely resembles how we express CSP instances, but
in constraint models we can have parameters. Effectively, constraint models express a
family of CSP instances.

A parameter is a variable that is set before the decision variables are set. The values
of the parameters determine the model instance (explained in more detail below), and
the parameters of a model is a way to mimic the parameters of the problem in question.
Note that parameters can also be expressed with respect to another parameter.

A decision variable is a similar concept to that of the CSP, although there are some
differences. Often, depending on the language, in constraint-based modelling decision
variables can have more advanced structures than that in the CSP. This includes arrays
and sets, for example. Furthermore, the variable domain of a decision variable may be
expressed with respect to some parameters.

A constraint is also a similar concept to that of the CSP. There are some differences
here as well; for example, a constraint may be expressed with respect to some parameters.
Moreover, in some cases it may be beneficial to have constraints whose removal does not
affect the solution space; such constraints are called implied constraints. Some solvers can
solve some models faster with the addition of implied constraints. In some cases it may
be beneficial to have constraints not specified in the problem that remove symmetric
solutions, and hence shrink the search space; such constraints are called symmetry-
breaking constraints. A symmetric solution is a solution that can be constructed in
polynomial time from another solution.

An objective function is similar to that of the COP. When modelling an optimisation
problem, an objective function is used. The objective function is expressed with respect
to parameters, or decision variables, or both. (In principle, the objective function can
be a constant function.)

It is important to differentiate between a problem instance and a model instance. A
problem instance is when the parameters of the problem are set, and a model instance
is, similarly, when the parameters of the model are set. For example, every set of clues
in the Sudoku problem constitutes a unique problem instance of the problem, and, given
a constraint model of the Sudoku problem, every possible combination of values of the
parameters of the model constitutes a unique model instance of the model. In this thesis
we will refer to both these concepts simply as instances, since it should follow from the
context which concept we are referring to, but it will be explicitly stated if there is a
possibility for misinterpretation or confusion.

2.4 MiniZinc

There are several solvers, many based on different solving technologies, for combinatorial
problems. Each solver usually has a solver-specific modelling language, and hence one
often needs to translate a model in order to test another solver or solving technology. The
MiniZinc language, a solver-independent high-level constraint-based modelling language,
and its toolchain were created in an attempt to unify different solving technologies and
solvers under a single modelling language [5, 6]. They have over the past few yearsecome increasingly popular. An important feature of the MiniZinc language is its rich
vocabulary, which allows modellers to write high-level declarative models that actually capture combinatorial substructures of a problem.

In Section 2.4.1, we describe the MiniZinc language along with a MiniZinc model of the BHPP that will be used throughout this thesis. In Section 2.4.2, we describe the MiniZinc toolchain and its features, including the FlatZinc language.

2.4.1 The MiniZinc Language

The MiniZinc language is a recurring theme throughout this thesis, mainly because we have implemented a tool that works (solely) on MiniZinc models, but also because the language is intuitive, rich, and well suited to base the theory on.

Recall the BHPP (black hole patience problem), for which we will present a MiniZinc model in order to explain the functionalities and syntax of the MiniZinc language. The model presented here is based on the auto-tabling model by Jip J. Dekker, which is based on the BHPP model from the MiniZinc Benchmark Suite.\footnote{Dekker's manually encapsulated version of the model can be found on \texttt{https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models} and the original model of the MiniZinc Benchmark Suite can be found on \texttt{https://github.com/MiniZinc/minizinc-benchmarks}. The model comes with a licence, which can be seen in Appendix A.}

Each declarative statement in MiniZinc is called an \textit{item}, some of which, those important for this thesis, are: include items, (parameter and decision) variable items, constraint items, solve items, and predicate items. Below, all of these are described.

In Figure 2.3, the include item of the BHPP model can be seen. An include item is used to include the information from another MiniZinc file into the current MiniZinc file. In this case, the contents of the file \texttt{"inverse.mzn"}, which consists of a predicate item defining the \textit{inverse} predicate used in the constraints presented below, are included.

In Figure 2.4, the parameter variable item of the BHPP model can be seen. In this case, the 17-by-3 \texttt{layout} array of integer parameters is declared. It is used to represent the distribution of the cards across the fans such that \texttt{layout[i,j]} refers to the \textit{j}th card in the \textit{i}th fan, where the Ace of Spades is represented by 1, the King of Spades by 13, the Ace of Hearts by 14, etc. The first card (\textit{j} = 1) of a fan denotes the top card of that fan. Parameter variable items declare the parameters in a MiniZinc model, which may be assigned (that is, the value of the parameter is set in the model) or, as in this case, unassigned (that is, the value of the parameter is set by the modeller when solving the model, effectively choosing an instance). These parameters can then be used throughout the model in expressions.

In Figure 2.5, the decision variable items of the BHPP model can be seen. Decision variable items declare the decision variables of the MiniZinc model. In this case, two 52-element arrays, \textit{x} and \textit{y}, of decision variables, both with domain \{1, 2, \ldots, 51, 52\},
array[1..17, 1..3] of int: layout; % Fans

Figure 2.4: The parameter variable item of the BHPP MiniZinc model.

array[1..52] of var 1..52: x; % Card at position
array[1..52] of var 1..52: y; % Position of card

Figure 2.5: The decision variable items of the BHPP MiniZinc model.

are declared. Both variables represent the same thing: the black hole. The value $x[i]$ denotes the card at $i$th position in the black hole, and the value $y[i]$ denotes the position of the card represented by $i$—where the first position denotes the bottom of the black hole. One reason for having two variables representing the same part of the problem is to enable better inference when solving, by allowing the use of different constraints that have complementary inference. Another reason is to get a more elegant model.

Note that what we call here parameter variable items and decision variable items are usually both simply referred to as variable items. However, to make the distinction clearer for the purposes of this thesis, we use this non-standard terminology.

MiniZinc supports, for decision variables and parameters, the following types: Booleans, integers, floats, strings, arrays of the aforementioned types, and sets of integers.

In Figure 2.6 we can see constraint items of the model. Constraint items express constraints of the MiniZinc model, and in this case there are four constraints.

The first constraint asserts that the first card in the black hole is the Ace of Spades. The second constraint asserts that, for every card in the black hole except the last one, the card above it is either one rank above or below, regardless of suit. By using the forall predicate over the parameter $i$ we can subtract the values of the cards and get the difference, and this difference should then be an element of the set expression in order for the constraint to be satisfied. Also note the domain annotation. This annotation is used to notify a constraint programming solver that domain consistency, when solving the expression preceding this annotation, should be used, if applicable. The third constraint is a channelling constraint, which makes sure that the decision variables $x$ and $y$ denote the same thing and that each of them benefits from the inference of the constraints on the other. The fourth and last constraint asserts that no card in any fan is played before the one above it (if any) in that fan.

In Figure 2.7 a solve item can be seen. The solve item of a MiniZinc model declares whether it is a minimisation, maximisation, or satisfaction problem that is modelled. In the case of a minimisation and maximisation, an objective function—which is expressed in terms of parameters and decision variables—is also defined. The solve item may take extra information, like in this case, called annotations, on how to look for a solution (e.g., branching techniques for the case of constraint programming).

In Figure 2.8 we can see the third constraint encapsulated within a predicate defi-
% Ace of Spades is first card
constraint x[1] == 1;

% Consecutive cards match
constraint forall (i in 1..51) ( ((x[i]-x[i+1]) in {13*j+1 | j in -4..3} union 
{13*k-1 | k in -3..4}) :: domain);

% Link x and y
constraint inverse(x, y) :: domain;

% A card must be played before the one under it.
constraint 
forall (i in 1..17 , j in 1..2) ( y[layout[i,j]] < y[layout[i,j+1]] );

Figure 2.6: The four constraints items of the BHPP MiniZinc model.

solve :: int_search(
   x,
   input_order,
   indomain_min,
   complete
) satisfy;

Figure 2.7: The solve item of the BHPP MiniZinc model.

nition, which is defined below it. Predicate definitions can be used as short-hands for
more complex expressions that may or may not depend on arguments.

The predicate definition is annotated with the auto-tabling annotation. The expres-
sions x[i] and x[i+1] have been put outside the predicate definition and are used as
arguments to denote a and b, respectively.

In Figure 2.9 the full model of the BHPP with the manually performed encapsulation
can be seen, and in Figure 2.10 the full model of the BHPP without the encapsulation
can be seen.

Note that MiniZinc supports several other items and functionalities; however, those
are deemed superfluous for this thesis.

2.4.2 The MiniZinc Toolchain and FlatZinc

The MiniZinc project offers more than just the MiniZinc language; it also includes the
MiniZinc toolchain, which has several useful tools and functionalities. The MiniZinc
% Consecutive cards match

constraint forall (i in 1..51) (
    adjacent(x[i], x[i+1])
);

predicate adjacent(var 1..52: a, var 1..52: b) :: presolve(autotable) =
    ((a-b) in {13*i+1 | i in -4..3} union {13*i -1 | i in -3..4}) :: domain;

Figure 2.8: The third constraint of the BHPP MiniZinc model encapsulated within a
predicate definition.

toolchain transforms a high-level richly expressed model into a language suitable for
solvers. This low-level language is called FlatZinc, and the solvers operate on Flat-
Zinc models. A MiniZinc model can be transformed into a FlatZinc model when given
values to all its parameters. The toolchain also has a user interface for writing MiniZinc
models.

FlatZinc is a sublanguage of MiniZinc; for example, a FlatZinc model does not have
parameters while a MiniZinc model does. As a result, a FlatZinc model has a direct
correlation to a CSP or COP instance.

2.5 Submodel Presolving with Auto-Tabling

We will revisit, and now explain further, the work of Jip J. Dekker et al. on auto-tabling
for submodel presolving in MiniZinc [7, 8]—which this thesis is mostly inspired by.

Submodel presolving is a well-known and powerful way to reformulate a constraint
model. Often it is done by putting an extensional constraint, with the presolved solutions
listed, in place of the submodel.

A simple way to separate a submodel in MiniZinc from the rest of the model is to
capsulate the submodel in a predicate definition. Since a predicate definition resembles
a standalone constraint model in itself, it makes presolving it a much simpler task. This
has been taken advantage of in the auto-tabling tool by Dekker [7, 8].

Dekker et al. introduce an approach to automate the process of presolving submodels
of a constraint model, and extend the MiniZinc toolchain to provide also an integrated
method for the presolving. The presolvable parts are limited to predicate definitions,
and a predicate definition can be selected for presolving via a new MiniZinc annotation.
The provided extension of the MiniZinc toolchain is shown to give runtime improvements
for several models and instances across multiple backends (technologies and solvers with
a MiniZinc interface). Furthermore, Dekker et al. argue that this approach is nonin-
trusive since the model remains readable and understandable, and that the approach

3More information about MiniZinc and its toolchain can be found on the MiniZinc homepage http://www.minizinc.org/
effectively moves the burden of presolving from the modeller to the modelling toolchain. Furthermore, many of the heuristics used in this thesis are inspired by or taken from the preliminary observations that Dekker et al. made regarding what constitutes a good predicate for presolving — namely the following observations: The predicate has few arguments; some argument is used more than once in the predicate definition; the predicate is used with a variable that is also used in the objective function; local decision variables are introduced in the predicate definition; and presolving the predicate definition results in a moderate number of solutions.
include "inverse.mzn";

% --- Parameters --- %
array[1..17, 1..3] of int: layout; % Fans

% --- Decision Variables --- %
array[1..52] of var 1..52: x; % Card at position
array[1..52] of var 1..52: y; % Position of card

% --- Constraints --- %
% Ace of Spades is first card
constraint x[1] == 1;

% Consecutive cards match
constraint forall (i in 1..51)(
  adjacent(x[i], x[i+1]) );

predicate adjacent(var 1..52: a,
  var 1..52: b) :: presolve(autotable) = ((a-b) in {13*i+1 | i in -4..3} union {13*i-1 | i in -3..4}) :: domain;

% Link x and y
constraint inverse(x, y) :: domain;

% A card must be played before the one under it.
constraint forall (i in 1..17, j in 1..2)(
  y[layout[i,j]] < y[layout[i,j+1]] );

% --- Solve --- %
solve :: int_search(
  x,
  input_order,
  indomain_min,
  complete
) satisfy;

Figure 2.9: The full BHPP MiniZinc model with the manually encapsulated submodel.
include "inverse.mzn";

% --- Parameters --- %
array[1..17, 1..3] of int: layout; % Fans

% --- Decision Variables --- %
array[1..52] of var 1..52: x; % Card at position
array[1..52] of var 1..52: y; % Position of card

% --- Constraints --- %
% Ace of Spades is first card
constraint x[1] == 1;

% Consecutive cards match
constraint forall (i in 1..51) ( ((x[i]-x[i+1]) in {13*j+1 | j in -4..3} union {13*k-1 | k in -3..4}) :: domain);

% Link x and y
constraint inverse(x, y) :: domain;

% A card must be played before the one under it.
constraint forall (i in 1..17, j in 1..2) ( y[layout[i,j]] < y[layout[i,j+1]] );

% --- Solve --- %
solve :: int_search(
   x,
   input_order,
   indomain_min,
   complete
) satisfy;

Figure 2.10: The full BHPP MiniZinc model with no encapsulation.
3 A Formalisation of the MiniZinc Language

A formal language for a subset of MiniZinc is presented. This formal language is called MiniZinc Logic (MZL) and is reminiscent of traditional first-order logics [11], but with some differences. MZL has four layers of interpretation—instead of one, like in traditional logics—and typed terms, to fully capture the structure of constraint-based modelling and MiniZinc in particular. In this logic there are no quantifiers, instead array comprehensions are used.

We present this formalised MiniZinc language to reason on, and define algorithms for, the MiniZinc language in a better and clearer manner. Note that MZL is just a formalisation of a subset of the MiniZinc language: it includes only the essential parts. However, the principles of this section still apply to the whole MiniZinc language.

Note that some non-standard terminology is used in this section due to the conflicts of terminology between logic and constraint modelling.

The syntax of MZL is defined and described in detail (Section 3.1). This includes the vocabulary and the grammar of the language. The semantics of MZL is then defined and described in detail (Section 3.2). This includes the type system and how the interpretations work. Lastly, we describe how the built-in functions of MiniZinc are represented in MZL (Section 3.3).

3.1 Syntax

We need a rigorous and understandable syntax in order to explain concisely and formally all the theory in this thesis. Directly using MiniZinc code and structures would not be good enough because it would be messy and hard to use that syntax and the whole of the MiniZinc language.

The syntax of MZL is similar to those of formal logics and is reminiscent of actual MiniZinc code. There are two parts of the syntax: the vocabulary and the grammar.

**Vocabulary**  An MZL vocabulary $V$ (or, simply, a vocabulary) is a set of symbols, which are divided first into different categories and then into classes. Every MZL vocabulary consists of two categories of symbols: names and signs.

The names are split into the following classes: constants, parameters, decision variables, argument variables, predicate symbols, and function symbols. Notice the difference from first-order logic, namely that the names in first-order logic are split into the classes: constants, variables, predicate symbols, and function symbols; however, the constants, parameters, and decision variables in MZL work similarly to the constants in first-order logic, and argument variables in MZL work similarly to the variables in first-order logic. The classes of the names are defined as follows:

- A *constant* is, in MZL, an integer—that is, any of the symbols from the set $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$—and is always interpreted as the same value (i.e., the symbol 1 is interpreted as the integer 1). Negative and multi-digit numbers are treated as one whole symbol in MZL. We assume that every vocabulary contains...
all the constants (one for each integer). We often use the meta-variable $c$ to denote an arbitrary constant.

- While a constant is always interpreted as the same value, a parameter may be interpreted to different values and can have more sophisticated types than integers—as explained in the semantics of MZL (Section 3.2). The values of the parameters determine the instance. We often use the meta-variable $n$ to denote an arbitrary parameter.

- A decision variable is very similar to a parameter, the difference being that a decision variable is defined and valuated at a later stage, and hence does determine the instance. Decision variables in MZL should not be confused with variables in first-order logics: decision variables are more similar to constants in first-order logics in that decision variables are interpreted to a value of its domain instead of ranging over its domain. We often use the meta-variable $x$ to denote an arbitrary decision variable.

- An argument variable is the counterpart, in MZL, of variables in first-order logics: an argument variables range over its domain. We assume that every vocabulary contains a countably infinite set of argument variables. The meta-variable $y$ is often used to denote an arbitrary argument variable.

- A predicate symbol is associated with a set of arguments and one or more truth values, since the truth value of a predicate call to the predicate symbol often depends on the values of the arguments provided in the predicate call. The meta-variable $P$ is often used to denote an arbitrary predicate symbol.

- A function symbol is associated with a set of arguments and a type, where the value of a function call to the function symbol often depends on the values of the arguments provided in the function call. We often use the meta-variable $f$ to denote an arbitrary function symbol.

The aforementioned meta-variable conventions include versions with prime marks and subscripts; hence, $n_1$, $n'$, and $n_{size}$ are all parameters.

The signs are split into two classes: connectives and punctuation. We assume that every vocabulary contains the whole set of signs listed below.

The following list constitutes the set of connectives of MZL:

- $\neg$ is the unary negation sign
- $\land$ is the binary conjunction sign  
- $\lor$ is the binary disjunction sign
- $\rightarrow$ is the binary implication sign  
- $\leftrightarrow$ is the binary equivalence sign
- $\bot$ is the nullary bottom sign  
- $\top$ is the nullary top sign

We use $\top$ and $\bot$ to respectively denote constant truth and constant falsehood across all interpretations.
The following list constitutes the set of punctuation signs of MZL:

( is the left parenthesis sign \(\) is the right parenthesis sign
[ is the left bracket sign ] is the right bracket sign
, is the comma sign : is the colon sign
| is the mid divider sign

Note that the equality symbol ‘=’ and inequality symbol ‘\(\neq\)’ are meta-relations that assert that two meta-variables refer to the same object or distinct objects, respectively. The ellipsis symbol ‘...’ is a meta-function, not part of MZL, used as an abbreviation for a long series of names. For example, \(f_1, \ldots, f_4\) represents the series \(f_1, f_2, f_3, f_4\). Also note the absence of quantifiers: this will be handled by array comprehensions, which are explained in the grammar paragraph below.

An MZL language \(L\) (or, simply, a language) over a vocabulary \(V\) consists of all the allowed constructions and interpretations possible from \(V\).

For the remainder of this thesis, we assume that \(L\) denotes an arbitrary language, except where explicitly stated.

**Grammar**  We have two different constructions in the MZL grammar: terms, and formulas. A term has one of many possible types and can be valuated to an element of its type, and a formula can be valuated only to either true or false.

Each term and formula can have a set of free argument variables, and every non-free argument variable is called bound. The function \(\text{free}\) maps every formula (and term) to its set of free argument variables. For example, let \(y\) and \(y'\) be argument variables of \(L\). Now, in the \(L\)-term \([f(y, y') \mid y]\), the argument variable \(y\) is bound but the argument variable \(y'\) is free.

To denote unary set union, we use \(\cup\); and given a set \(S\) of sets, it is defined as:

\[ x \in \bigcup S \iff \exists X \in S, \ x \in X. \]

**Definition 3.1 (L-Terms).** Let \(k\) range over the positive integers \(\mathbb{Z}_+\). The set of \(L\)-terms and the function \(\text{free}\) over \(L\)-terms are inductively defined as follows:

- Every constant \(c\), parameter \(n\), decision variable \(x\), and argument variable \(y\) of \(L\) is an \(L\)-term. Also:
  \[
  \text{free}(c) = \text{free}(n) = \text{free}(x) = \emptyset, \\
  \text{free}(y) = \{y\}.
  \]

- If \(\tau_1, \ldots, \tau_n\) are \(L\)-terms, then, for every \(L\)-term \(\tau\), the expression \(\tau[\tau_1, \ldots, \tau_k]\) is an \(L\)-term. Also:
  \[
  \text{free}(\tau[\tau_1, \ldots, \tau_k]) = \text{free}(\tau) \cup \bigcup_{i=1}^k \text{free}(\tau_i).
  \]
• If \( \tau_1, \ldots, \tau_k \) are \( L \)-terms, then, for every function symbol \( f \) of \( L \), the expression \( f(\tau_1, \ldots, \tau_k) \) is an \( L \)-term. Also:

\[
\text{free}(f(\tau_1, \ldots, \tau_k)) = \bigcup_{i=1}^{k} \text{free}(\tau_i).
\]

• If \( \tau_1, \ldots, \tau_k \) are \( L \)-terms, then \([\tau_1, \ldots, \tau_k] \) is an \( L \)-term. Also:

\[
\text{free}([\tau_1, \ldots, \tau_k]) = \bigcup_{i=1}^{k} \text{free}(\tau_i).
\]

• If \( \tau \) is an \( L \)-term, \( y_1, \ldots, y_k \) are argument variables of \( L \), and \( \varphi \) is an \( L \)-formula (defined below), then the array comprehensions \([\tau \mid y_1], [\tau \mid y_1, \ldots, y_k], [\tau \mid y_1 : \varphi], [\tau \mid y_1, \ldots, y_k : \varphi] \) are all \( L \)-terms. Also:

\[
\begin{align*}
\text{free}([\tau \mid y_1]) &= \text{free}([\tau \mid y_1 : \varphi]) = \text{free}(\tau) \setminus \{y_1\}, \\
\text{free}([\tau \mid y_1, \ldots, y_k]) &= \text{free}([\tau \mid y_1, \ldots, y_k : \varphi]) = \text{free}(\tau) \setminus \{y_1, \ldots, y_k\}.
\end{align*}
\]

• If \( \varphi \) is an \( L \)-formula, then \( f_{\text{bool2int}}(\varphi) \) is an \( L \)-term, where \( f_{\text{bool2int}} \) is a special base language function (explained in Section 3.3). Also:

\[
\text{free}(f_{\text{bool2int}}(\varphi)) = \text{free}(\varphi).
\]

• If \( \tau \) is an \( L \)-term and \( y_1, \ldots, y_k \) are argument variables, then the let-expression \( y_1, \ldots, y_k : \tau \) is an \( L \)-term. Also:

\[
\text{free}(y_1, \ldots, y_k : \tau) = \text{free}(\tau).
\]

□

**Definition 3.2 (L-Formulas).** Let \( k \) range over the positive integers \( \mathbb{Z}_+ \). The set of \( L \)-formulas and the function \( \text{free} \) over \( L \)-formulas are inductively defined as follows:

• Bottom \((\bot)\) and top \((\top)\) are \( L \)-formulas. Also:

\[
\text{free}(\bot) = \text{free}(\top) = \emptyset.
\]

• If \( \tau_1, \ldots, \tau_n \) are \( L \)-terms, then, for every predicate symbol \( P \) of \( L \), the expression \( P(\tau_1, \ldots, \tau_k) \) is an \( L \)-formula. Also:

\[
\text{free}(P(\tau_1, \ldots, \tau_k)) = \bigcup_{i=1}^{k} \text{free}(\tau_i).
\]

• If \( \varphi \) is an \( L \)-formula, then \( \neg \varphi \) is an \( L \)-formula. Also:

\[
\text{free}(\neg \varphi) = \text{free}(\varphi).
\]

27
• If ϕ and ψ are \(L\)-formulas, then \(ϕ ∧ ψ\) is an \(L\)-formula. Also:
\[
\text{free}(ϕ ∧ ψ) = \text{free}(ϕ) \cup \text{free}(ψ).
\]

The construction for every other binary connective is analogous.

Often, when the properties of the language \(L\) do not contribute to the discussion, we will not specify the language and simply say that \(ϕ\) is a formula, that \(τ\) is a term, that \(y\) is an argument variable, and so on.

A term or formula is called open if it contains at least one free argument variable, and called closed otherwise.

To form a better understanding of how this syntax relates to MiniZinc, we will show how the BHPP model (shown in Figure 2.10) would look like in MZL syntax. When showing parts of the BHPP MiniZinc model, for reference, annotations and other parts of the language that are not supported in MZL are removed for clarity.

Starting with the vocabulary, let \(\text{layout}\) be denoted by the parameter \(l\), and let \(x\) and \(y\) be denoted, respectively, by the two decision variables \(x\) and \(y\).

The first constraint (\(\text{constraint } x[1] == 1\)) is represented by the formula \(P_=(x[1], 1)\). Here the predicate symbol \(P_=(\text{in})\) denotes the base language predicate for equality.

The second constraint is more complicated, and since we have no sets in MZL (for simplicity) the set expression is denoted by list expression. Starting with the list expression, the MiniZinc expression
\[
\{13*i + 1 \mid i \text{ in } -4..3\} \text{ union } \{13*i - 1 \mid i \text{ in } -3..4\}
\]
is represented by the term
\[
\text{f\textsubscript{++}}([\text{f\textsubscript{+}}(13, j), 1] \mid j), [\text{f\textsubscript{+}}(13, j), 1] \mid k]).
\]
Here, the function symbol \(\text{f\textsubscript{++}}\) is the base language function that concatenates two lists into one (and acts in place of the \(\text{union}\) in this case), the function symbol \(\text{f\textsubscript{+}}\) is the base language function that denotes addition, and \(\text{f\textsubscript{*}}\) is the base language function that denotes multiplication. In the list comprehensions, \(j\) denotes \(j\) and \(k\) denotes \(k\). The whole constraint, using \(τ\) to denote the list expression above, is
\[
\text{constraint forall } (i \text{ in } 1..51) ((x[i] - x[i+1]) \in τ)
\]
and is represented by the formula
\[
P_\forall([\text{f\textsubscript{bool2int}}(P_\text{in}(x[i] - x[i+1], τ)) \mid i]).
\]
Here, the predicate symbol \(P_\text{in}\) is a base language predicate and is true if the value of the first argument is an element of the value of the second argument, which is a list. And the predicate symbol \(P_\forall\) is the base language predicate that denotes the \(\text{forall}\) predicate. Note that the base language function \(\text{f\textsubscript{bool2int}}\), which converts \(\text{true}\) to 1 and \(\text{false}\) to 0, is used because MZL has no automatic coercion like MiniZinc has.
The third constraint \((\text{constraint inverse}(x, y))\) is simply be represented by the formula \(P_{\text{inverse}}(x, y)\), where the predicate symbol \(P_{\text{inverse}}\) denotes the base language predicate that corresponds to the \text{inverse} global MiniZinc constraint.

The fourth and final constraint

\[
\text{constraint forall} \ (i \ \text{in} \ 1..17, \ j \ \text{in} \ 1..2) \ (y[\text{layout}[i,j]] < y[\text{layout}[i,j+1]])
\]

is represented by the formula

\[
P_Y[f_{\text{bool2int}}(P < (y[l[a,b]], y[l[a,f(b,1)]]))] | a, b].
\]

In the list comprehension, \(a\) denotes \(i\) and \(b\) denotes \(j\).

### 3.2 Semantics

The semantics is defined mainly in order to make terms have types, since types are important for the theory and heuristics of finding profitably presolvable submodels. We also explain how interpretations work. The semantics is incomplete; however, this is not a problem since the absent parts are not necessary for our purposes.

There are four layers of interpretations in MZL: declarations, definitions, instantiations, and assignments.

First, an \(L\)-declaration gives a type (or a type signature, in the case of predicate and function symbols) to all the names of the language.

Secondly, an \(L\)-definition gives meaning to the function symbols and predicate symbols, such that they can be valued. It also defines whether the goal of the model is satisfaction or optimisation, and in the latter case defines an objective function as well.

Thirdly, an \(L\)-instantiation fixes all parameters and creates an instance of the model.

Finally, an \(L\)-assignment fixes all the decision variables of the model. Given a set of constraints (i.e., formulas), an \(L\)-assignment either is a solution or violates some of the constraints.

There is a direct correspondence between a MiniZinc model and, together, an \(L\)-declaration, an \(L\)-definition, and a set of \(L\)-formulas (the constraints).

#### Types

No thorough type system is presented for MZL in this thesis because it is deemed too time consuming to design and not fully necessary for the results. However, we will present a partly formal and otherwise informal description of a type system.

Every constant, parameter, decision variable, and argument variable is given a type. Every predicate symbol and function symbol is given a type signature. Every term has a type, which is derived from the names used in and the structure of the term itself.

The classes are \texttt{con}, \texttt{par}, and \texttt{var}.

**Definition 3.3 (Classes).** The \textit{class} of a term or formula represents where in the process the term or formula is fully interpreted. There are three classes and they are \texttt{con} (fully interpreted at definition level), \texttt{par} (fully interpreted at instantiation level), and \texttt{var} (not fully interpreted, only partially in terms of a constraint).
The class of a term or formula is determined at declaration level. There is a hierarchy of the classes: all con terms are also par terms, and all par terms are also var terms. The hierarchy is analogous for formulas. We say that con is a subclass of par and that par is a subclass of var, and every class is a subclass of itself. Given a set of terms $T$ or a set of formulas $\Phi$, the supreme class of $T$ (or $\Phi$) is the least class to which the class of each term in $T$ (or formula in $\Phi$) is a subclass.

As a result of the levels of interpretation of the classes, all constants are con, all parameters are par, all decision variables are var, each argument variable is either par or var, and each of the rest of the terms is any one of them, determined by deriving the class from the components of the term. The classes of formulas are derived in a similar manner.

For convenience, every term is represented as an array. The dimensionality of an array determines if it is, for example, a one-dimensional array, two-dimensional array (i.e., a matrix), or a zero-dimensional array (i.e., a single value). The type of a term, defined next, is essentially a specification of the class, the array dimensions, and the domain of the term:

**Definition 3.4 (Types).** The type $\sigma$ of a term $\tau$ consists of three parts: its class, its domain (which is a subset of $\mathbb{Z}$ and is denoted $\text{dom}(\tau)$), and its dimension set \{\tau_1, \tau_2, \ldots, \tau_k\} (denoted $\text{dim}(\tau)$) of con terms, called dimension terms, and where $k$ is called the dimensionality of $\tau$; we say that $\tau$ is $k$-dimensional. The terms $\tau_1, \tau_2, \ldots, \tau_k$ must be 0-dimensional. We use an indexing convention starting from 1.

Note that the domain and the dimensionality of a term are invariable across instances, but that the dimension terms may vary across instances. Circular dependencies are not allowed, meaning, for example, that if a dimension term of $\tau$ is $\tau'$, then $\tau'$ cannot have $\tau$ as a dimension term. Also note that if one of the dimension terms valuates to 0 in some interpretation, then the array is empty, meaning it has no elements.

Some conventions are in place: every constant must be 0-dimensional, and the constant 1 has \{1\} as domain, the constant 2 has \{2\} as domain, and so on.

How these types are used and what they mean will be clearer after $L$-definitions and derived types have been defined and explained below. First we will revisit the MZL translation (of the end of Section 3.1) of the BHPP MiniZinc model (of Figure 2.10). In order to make the semantics of the MiniZinc model correspond to the semantics of our MZL translation, we have an $L$-declaration $D$ that satisfies the following conditions. The parameter $l$, which represents the parameter variable item array [1..17, 1..3] of int: layout, is a 2-dimensional par parameter with domain $\mathbb{Z}$ and dimension set \{17, 3\}, where 17 and 3 are two 0-dimensional con constants. The decision variables $x$ and $y$, which respectively represent the decision variable items array [1..52] of var 1..52: $x$ and array [1..52] of var 1..52: $y$, are both 1-dimensional

\footnote{Note that in MiniZinc, as opposed to here, the domain of a type may be equivalently expressed as a par term. Including par terms in this case makes the type system more complicated than necessary for applying the theory of this thesis to MiniZinc.}
**var** decision variables with domain \(\{1, \ldots, 52\}\) and dimension set \(\{52\}\), where 52 is a 0-dimensional **con** constant.

We also have a concept of subtypes. Note that every type is a subtype of itself.

**Definition 3.5** (Subtypes). Given two types \(\sigma\) and \(\sigma'\), the latter is a **subtype** of the former if and only if the three following conditions are fulfilled:

- The class of \(\sigma'\) is a subclass of the class of \(\sigma\).
- The domain of \(\sigma'\) is a (possibly non-strict) subset of the domain of \(\sigma\).
- \(\sigma\) and \(\sigma'\) have the same dimension set. \(\Box\)

A function or predicate symbol does not have a type; it has a type signature instead.

**Definition 3.6** (Type Signature). Let \(k\) range over the non-negative integers \(\mathbb{N}\), and let \(\sigma, \sigma_1, \ldots, \sigma_k\) be types.

A **type signature** \(\sigma_1 \times \cdots \times \sigma_k \rightarrow \sigma\) is the type signature for a \(k\)-ary function with return type \(\sigma\) and where the \(i\)th function argument has type \(\sigma_i\), for \(1 \leq i \leq k\).

A type signature \(\sigma_1 \times \cdots \times \sigma_k\) is the type signature for a \(k\)-ary predicate where the \(i\)th predicate argument has type \(\sigma_i\), for \(1 \leq i \leq k\). \(\Box\)

An \(L\)-declaration associates each name with a type or type signature:

**Definition 3.7** (**L-Declarations**). Let \(k\) range over the non-negative integers \(\mathbb{N}\). An \(L\)-declaration \(D\) defines the type or type signature of every name \(a\) of MZL, denoted \(a^D\), such that:

- Every constant \(c\) gets associated with the 0-dimensional **con** type with the singleton domain of \(c\) (e.g, \(\text{dom}(1^D) = \{1\}\)).

- Each parameter \(n\) gets associated with a **par** type \(n^D\).

- Each decision variable \(x\) gets associated with a **var** type \(x^D\).

- Each argument variable \(y\) gets associated with a type \(y^D\), which, depending on the \(L\)-declarations, is a **par** or **var** type.

- Each function symbol \(f\) gets associated with an arity \(k\); a type signature \(\sigma_1 \times \cdots \times \sigma_k \rightarrow \sigma\), where \(\sigma, \sigma_1, \ldots, \sigma_k\) are types; and \(k\) argument variables \(y_1, \ldots, y_k\), called the **arguments** of \(f^D\); such that \(y_1^D = \sigma_1, \ldots, y_k^D = \sigma_k\). The class of \(\sigma\) must be the supreme class of the set \(\{y_1, \ldots, y_k\}\).

- Each predicate symbol \(P\) gets associated with an arity \(k\); a type signature \(\sigma_1 \times \cdots \times \sigma_k\), where \(\sigma, \sigma_1, \ldots, \sigma_k\) are types; and \(k\) argument variables \(y_1, \ldots, y_k\), called the arguments of \(P^D\); such that \(y_1^D = \sigma_1, \ldots, y_k^D = \sigma_k\). \(\Box\)
As a result, in each $L$-declaration, a term either is associated with a type, which is derivable from the $L$-declaration, or has no derivable type. Terms for which the former is true are called type-correct; and terms for which the latter is true are called type-erroneous.

**Definition 3.8** (Derived Types and Type-correct Terms). Let $k$ range over the non-negative integers $\mathbb{N}$, and let $D$ be an $L$-declaration. Every term for which no derived type is defined is a type-erroneous term; and if a term has a derived type, then it is type-correct. The derived types of the set of all terms are inductively defined as follows:

- If $\tau_1, \ldots, \tau_k$ are 0-dimensional terms and $\tau$ is a $k$-dimensional term, then $\tau[\tau_1, \ldots, \tau_k]$ is a 0-dimensional term with domain $\text{dom}(\tau^D)$. The class of $\tau[\tau_1, \ldots, \tau_k]$ is the supreme class of the set $\{\tau, \tau_1, \ldots, \tau_k\}$.

- If $\tau_1, \ldots, \tau_k$ are terms and $f$ is a function symbol with type $\sigma_1 \times \cdots \times \sigma_k \rightarrow \sigma$, such that, for $1 \leq i \leq k$, the type $\tau_i^D$ is a subtype of $\sigma_i$, the type $\sigma'$ is a subtype of $\sigma$, and the supreme class of the set $\{\tau_1, \ldots, \tau_k\}$ is the class of $\sigma'$, then the $L$-term $f(\tau_1, \ldots, \tau_k)$ has the type $\tau'$.

- If $\tau_1, \ldots, \tau_k$ are 0-dimensional $L$-terms, then $[\tau_1, \ldots, \tau_k]$ is a 1-dimensional $L$-term with domain $\bigcup_{i=1}^{k} \text{dom}(\tau_i^D)$. The class of $[\tau_1, \ldots, \tau_k]$ is the class of $\{\tau_1, \ldots, \tau_k\}$.

- If $\tau$ is a 0-dimensional $L$-term, the argument variable $y$ is 0-dimensional, and $\varphi$ is an $L$-formula such that $\text{free}(\varphi) \subseteq \{x\}$, then $[\tau \mid x]$ and $[\tau \mid x : \varphi]$ are both 1-dimensional $L$-terms with domain $\text{dom}(\tau^D)$ and the same class as $\tau$.

- If $\tau$ is a 0-dimensional $L$-term, the argument variables $y_1, \ldots, y_k$ are 0-dimensional, and $\varphi$ is an $L$-formula such that $\text{free}(\varphi) \subseteq \{y_1, \ldots, y_k\}$, then $[\tau \mid y_1, \ldots, y_k]$ and $[\tau \mid y_1, \ldots, y_k : \varphi]$ are both 1-dimensional $L$-terms with domain $\text{dom}(\tau^D)$ and the same class as $\tau$.

- If $\varphi$ is a formula, then $f_{\text{bool2int}}(\varphi)$ is a 0-dimensional $L$-term with domain $\{0, 1\}$ and the same class as $\varphi$.

If a term $\tau$ is type-correct under an $L$-declaration $D$, then we write $D \models \tau$.

The notion of type correctness is extended to $L$-formulas:

**Definition 3.9** (Derived Class and Type-correct $L$-Formulas). Let $k$ range over the non-negative integers $\mathbb{N}$, and let $D$ be an $L$-declaration. Every term for which no derived type is defined is a type-erroneous term. The set of type-correct formulas are inductively defined as follows:

- $\top$ and $\bot$ are type correct $\text{con}$ formulas.

- If $\tau_1, \ldots, \tau_k$ are terms and $P$ is a predicate symbol with type $\sigma_1 \times \cdots \times \sigma_k$, such that, for $1 \leq i \leq k$, the type $\tau_i^D$ is a subtype of $\sigma_i$, then the formula $P(\tau_1, \ldots, \tau_k)$ is a type-correct formula, where the supreme class of the set $\{\tau_1, \ldots, \tau_k\}$ is the class of this formula.
• If \( \phi \) is type correct, then \( \neg \phi \) is a type-correct formula with the same class.

• If \( \phi \) and \( \psi \) are type correct formulas, then \( \phi \land \psi \) is a type-correct formula with the same class.

The rule for every other binary connective is analogous.

Like with terms, if a formula \( \phi \) is type-correct under an \( L \)-declaration \( D \), then we write \( D \models \phi \).

For the remainder of this thesis, we assume that all concerned terms and formulas are type-correct, and simply disregard incorrect ones. The analysis of that distinction is irrelevant for this thesis.

**L-Definitions** In an \( L \)-definition, the function symbols and predicate symbols are defined, and, optionally, an objective term (objective function) is also defined.

Every definition of function symbols (called *function definitions*) and predicate symbols (called *predicate definitions*) is either atomic or composite. A *composite* function definition associates a function symbol with a term; a composite predicate definition associates a predicate symbol with a formula; an *atomic* function definition associates a function symbol with a mathematical function; and an atomic predicate definition associates a predicate symbol with a mathematical relation. This distinction is used to mimic more closely the MiniZinc language.

Atomic function and predicate definitions will be used to mimic the built-in functions and predicates of the MiniZinc language, for example addition and equality, but also the global constraints.\(^5\) Composite function and predicate definitions will be used to mimic function and predicate definitions specified in a MiniZinc model.

**Definition 3.10 (L-Definitions).** Given an \( L \)-declaration \( D \), an \( L \)-definition \( E \) does the following:

• It associates every function symbol \( f \) of \( L \) with a term \( \tau \) (or a function \( f \), depending on the \( L \)-definition) such that free(\( \tau \)) is a subset of the set of arguments of \( f^D \) (or the arity and type of \( f \) match the arity and type of \( f^D \)). We say that \( f \) is *defined* as \( \tau \) (or \( f \)) under \( E \).

• It associates every predicate symbol \( P \) of \( L \) with an \( L \)-formula \( \varphi \) (or a relation \( R \), depending on the \( L \)-definition) such that free(\( \varphi \)) is a subset of the set of arguments of \( P^D \) (or the arity and type of \( R \) match the arity and type of \( P^D \)). We say that \( P \) is *defined* as \( \varphi \) (or \( R \)) under \( E \).

• Optionally, it defines an *objective term* \( \tau \), which is used for optimisation problems and is a closed term. We say that \( E \) *has* an objective term \( \tau \). The goal is always to minimise the objective term, without loss of generality; for maximisation problems, negated values are used instead.

\(^5\) In MiniZinc, whether a global constraint is, what is here called, composite or atomic depends on the solver used.
An \(L\)-constraint is a special formula that is used as one of the constraints of a model:

**Definition 3.11.** An \(L\)-constraint \(\varphi\) is a type-correct closed \(L\)-formula that determines whether an instantiation is valid and whether an assignment is a solution. An \(L\)-constraint set is a set \(\Phi\) of \(L\)-constraints. An \(L\)-declaration and an \(L\)-definition together with an \(L\)-constraint set form an \(L\)-model.

For example, the MZL formulas presented at the end of Section 3.1 form the \(L\)-constraint set of the MZL model of the presented MiniZinc model of the BHPP.

**\(L\)-Instantiations** An \(L\)-instantiation is the act of fixing the parameters of the language \(L\):

**Definition 3.12 (\(L\)-Instantiations).** Given an \(L\)-declaration \(D\) and an \(L\)-definition \(E\), an \(L\)-instantiation \(I\) does the following: For every parameter and \texttt{par} argument variable \(a\), the interpretation \(a^I\) is equated with an element of the domain \(\text{dom}(a^D)\).

An \(L\)-model together with an \(L\)-instantiation form an \(L\)-instance of the \(L\)-model, or simply called an \(L\)-instance.

The number of \(L\)-instantiations, and hence \(L\)-instances, is often infinite. The set of all \(L\)-instances is called the \textit{instance universe}.

At this stage, every \texttt{par} term and \texttt{par} formula can be valuated. Given an \(L\)-declaration \(D\) and an \(L\)-definition \(E\), if a \texttt{par} formula \(\varphi\) is \texttt{true} under an \(L\)-instantiation \(I\), then we write \(D, E, I \models \varphi\).

**\(L\)-Assignments** An \(L\)-assignment is the interpretation that is applied last. In practice, this interpretation is performed by a solver.

**Definition 3.13 (\(L\)-Assignments).** Given an \(L\)-declaration \(D\), an \(L\)-definition \(E\), and an \(L\)-instantiation \(I\), an \(L\)-assignment \(A\) does the following: For every decision variable and \texttt{var} argument variable \(a\) of \(L\), the interpretation \(a^A\) is equated with an element of the domain of \(a^D\).

The number of \(L\)-assignments is often very large, especially when dealing with \textit{NP}-hard problems. The set of all possible \(L\)-assignments is called the \textit{search space} of the \(L\)-instance. Since we are dealing with combinatorial problems, the search space is here always discrete.

At this stage, every \texttt{var} term and \texttt{var} formula can be valuated. Given an \(L\)-assignment, every \texttt{var} term can be valuated to an element of its domain, and every \texttt{var} formula can be valuated to either \texttt{true} or \texttt{false}.

Given an \(L\)-declaration \(D\), an \(L\)-definition \(E\), and an \(L\)-instantiation \(I\), if a \texttt{var} formula \(\varphi\) is \texttt{true} under an \(L\)-assignment \(A\), then we write \(D, E, I, A \models \varphi\).

If all formulas of a constraint set are valuated to true, then the \(L\)-assignment is a \textit{solution} to the \(L\)-instance, else it is a \textit{non-solution}. The set of all solutions is called the \textit{solution space} of the \(L\)-instance.
If the objective $L$-term is optimal (i.e., minimal), then it is an optimal solution to the instance.

If an instance has a solution, then it is called satisfiable, else unsatisfiable.

### 3.3 Base Language: Atomic Functions and Predicates

In MiniZinc there is a set of predicates and functions that are built-in to the language. In other words, each of these predicates and functions have the same meaning, and can be used, across all MiniZinc models. This set includes predicates for equality, inequality, and disequality; and functions for addition, subtraction, and multiplication.

To mimic this functionality, we have some reserved predicate symbols and function symbols in MZL, as seen throughout Sections 3.1 and 3.2. These are called base language functions and base language predicates, and they have the same type across all $L$-declarations and the same definition across all $L$-definitions.

In an equal fashion, there is a set of predicates that are so-called global constraints. For example, $P_{\text{alldifferent}}$ is the all-different global constraint [4].

The special atomic function $f_{\text{bool2int}}$, taking a formula as argument, represents 1 if the formula is valuated to true, and 0 if it is valuated to false. This function acts as the Iverson bracket in MZL.

Some interesting atomic predicates are $P_{\forall}$ and $P_{\exists}$. These predicates are used instead of universal and existential quantification. The $P_{\forall}$ predicate takes a 1-dimensional var array with $\{0, 1\}$ as domain, and will represent true if and only if all elements of the array are valuated to 1. The $P_{\exists}$ predicate is defined analogously, but it will only represent false if all elements of the array are valuated to 0.

### 4 Theory and Design

Using the formal language MZL from Section 3, we detail algorithms for detecting potentially profitably presolvable submodels of a model and encapsulating such submodels in a predicate for presolving.

In Section 4.1, we present our method, along with the necessary concepts, for identifying all (or a subset of) the submodels present in a MiniZinc model. This method uses a structure that captures the submodels. The encapsulating process is explained in Section 4.2, which uses the same structure as the identification process to infer how to rewrite the model. The submodels are ranked, as explained in Section 4.3, using a set of heuristics, so that the submodel with the highest confidence of profitability of presolving is chosen when encapsulating.

#### 4.1 Submodel Identification

We introduce and describe a structure that will be used to identify submodels in a model and to, as explained later, encapsulate them properly and automatically. This structure is called a submodel identification graph (SIG), which is a directed acyclic graph (DAG) where each vertex represents a submodel of the model. A DAG is a graph
\( G = (V, E) \), where \( V \) is a set of vertices and \( E \) is a set of directed edges between the vertices, \( E \subseteq \{(u, u') \mid u, u' \in V\} \), such that there are no cycles in \( G \). Each edge of a SIG defines a relationship between two submodels (represented by its incident vertices) and all these edges are necessary for encapsulating a selected submodel via the SIG. The process of constructing a SIG from a model is called submodel identification.

Two formulas can be related in three different, and mutually exclusive, ways: One can be a subformula of the other (Section 4.1.1), an abstraction of the other (Section 4.1.2), or a generalisation of the other (Section 4.1.3). All of these relations are partial orders over submodels, and, since they are mutually exclusive, disjunctively combining them produces a new partial order. A DAG can be used to represent a partial order \( \preceq \) over a set \( A \): each element of \( A \) is represented as a vertex in the DAG and \( a \preceq b \) if and only if \( a \) is reachable from \( b \), for every \( a, b \in A \), in the DAG. This is how a SIG represents the relations mentioned above between a set of submodels. Finally, everything is tied together and the concept of SIGs is formally defined (Section 4.1.4).

First, we will give examples of a subformula, an abstraction, and a generalisation, purely in MiniZinc syntax, to give the reader a better understanding of the concepts. Consider the second constraint of the BHPP model (of Section 2.4.1) which, without the annotations and the ‘constraint’ keyword, is

\[
\forall i \in 1..51 ( (x[i]-x[i+1]) \in \{13*j+1 \mid j \in -4..3\} \cup \{13*k-1 \mid k \in -3..4\})
\]

A subformula of the formula above is

\[
((x[i2]-x[i2+1]) \in \{13*j+1 \mid j \in -4..3\} \cup \{13*k-1 \mid k \in -3..4\})
\]

where \( i2 \) is a fresh parameter that represents \( i \) from the original formula and has the parameter variable item \( 1..51: i2 \) (the domain is derived from \( i \)). An abstraction of the subformula above is

\[
(a-b) \in \{13*j+1 \mid j \in -4..3\} \cup \{13*k-1 \mid k \in -3..4\}
\]

where \( a \) and \( b \) are fresh decision variables that respectively represent \( x[i2] \) and \( x[i2+1] \) and are respectively defined by the decision variable items \( \text{var 1..52: a and var 1..52: b} \) (the domains are derived from \( x \)). By replacing a term by a parameter or decision variable that represent that term, we make the formula more abstract. A generalisation of the subformula above is

\[
((x[i2]-x[i3+1]) \in \{13*j+1 \mid j \in -4..3\} \cup \{13*k-1 \mid k \in -3..4\})
\]

where \( i3 \) is a fresh parameter that generalises the second \( i2 \) to a parameter of the parameter variable item \( \text{int: i3} \) (note the enlargement of the domain and the separation of the first \( i2 \) from the second, making the whole formula more general).
4.1.1 Subformulas

As seen in the grammar of terms and formulas, the two are either constructed from smaller terms or formulas, or are atomic (e.g., constants). Many a formula or term is constructed from other formulas and terms, which are the subformulas and the subterms of that formula or term. The concept of subformulas is useful when identifying submodels because every subformula of the constraints is, in fact, a submodel. In order to define subformulas of formulas, it is necessary to define both subformulas and subterms of both formulas and terms.

Consider, again, the third constraint of the BHPP MiniZinc model (shown in Figure 2.6). It is

\[
\textsf{constraint forall}(i \in 1..51) \ ( (x[i]-x[i+1]) \ \mathbf{in} \ \{13*j+1 \mid j \ \mathbf{in} \ -4..3\} \ \mathbf{union} \ \{13*k-1 \mid k \ \mathbf{in} \ -3..4\} : : \ \textsf{domain});
\]

and recall (from the end of Section 3.1) that, in MZL, it is represented by the formula

\[
\text{P} \forall (\text{bool2int}(\text{P} \in (x[i] - x[i + 1], \tau))) \mid i],
\]

where \(\tau\) is the term

\[
f_{++}(f_{*}(13,j),1) \mid j], [f_{*}(f_{*}(13,k),1) \mid k]).
\]

We will refer to the formula as \(\phi\). The subformulas of \(\phi\) are \(\phi\) itself and the formula

\[
\text{P}_{\text{in}}(x[i] - x[i + 1], \tau),
\]

where i is now a free argument variable and represents the MiniZinc expression

\[
(x[i2]-x[i2+1]) \ \mathbf{in} \ \{13*j+1 \mid j \ \mathbf{in} \ -4..3\} \ \mathbf{union} \ \{13*k-1 \mid k \ \mathbf{in} \ -3..4\}
\]

where \(i2\) is a fresh parameter defined by the implicit parameter variable item 1..51: i2, which follows from the declaration in the \textsf{forall} generator \(i \mathbf{in} 1..51\). This formula is a subformula of \(\phi\) since it is a formula and \(\phi\) is constructed from it. Note that, in this case, in MiniZinc a fresh parameter had to be added since \(i\) is only defined within the scope of the \textsf{forall} expression, and in this case the \textsf{forall} expression and its associated scope are no longer present. Note that the MiniZinc expression is similar to the expression that is inside the predicate definition of the manually encapsulated submodel listed in Figure 2.8.

Similarly, the subterms of \(\phi\) can be identified, which there are a lot more of in this case. For example, we have the MiniZinc expression 13*j2+1, where \(j2\) is defined by the implicit parameter variable item -4..3: j2. This expression is represented by the MZL term \(f_{*}(f_{*}(13,j),1)\), which is a subterm of \(\phi\).

\textbf{Definition 4.1 (Subformulas and Subterms of Terms).} Let \(k\) range over the non-negative integers \(\mathbb{N}\). Subformulas and subterms of terms are inductively defined as follows:
• Every term is a subterm of itself.
• Every constant, parameter, decision variable, and argument variable has no subterms.
• If \( \tau_1, \ldots, \tau_k \) are terms, then, for every term \( \tau \), the terms \( \tau_1, \ldots, \tau_k \) are subterms of \( \tau[\tau_1, \ldots, \tau_k] \).
• If \( \tau_1, \ldots, \tau_k \) are terms, then, for every function symbol \( f \), the terms \( \tau_1, \ldots, \tau_k \) are subterms of \( f(\tau_1, \ldots, \tau_k) \).
• If \( \tau_1, \ldots, \tau_k \) are terms, then the terms \( \tau_1, \ldots, \tau_k \) are subterms of \( [\tau_1, \ldots, \tau_k] \).
• If \( \tau \) is a term, \( y_1, \ldots, y_k \) are argument variables, and \( \varphi \) is a formula, then \( \tau \) and \( y_1 \) are subterms of \( [\tau \mid y_1] \) and \( [\tau \mid y_1 : \varphi] \) (with \( \varphi \) as a subformula, defined next, of the latter), and \( \tau, y_1, \ldots, y_k \) are subterms of \( [\tau \mid y_1, \ldots, y_k] \) and \( [\tau \mid y_1, \ldots, y_k : \varphi] \) (with, again, the formula \( \varphi \) as a subformula of the latter).
• If \( \varphi \) is a formula, then \( \varphi \) is a subformula of the term \( \text{fbool2int}(\varphi) \).
• If \( \tau \) is a term and \( \varphi \) is a subformula of \( \tau \), then every subterm (and subformula) of \( \varphi \) is a subterm (or subformula) of \( \tau \).
• If \( \tau' \) is a subterm of \( \tau \), then every subterm (and subformula) of \( \tau' \) is also a subterm (or subformula) of \( \tau \).  

The subterm relation is antisymmetric, transitive, and reflexive over terms; hence it is a partial order over terms.

**Definition 4.2** (Subformulas and Subterms of Formulas). Let \( E \) be an \( L \)-definition and let \( k \) range over the non-negative integers \( \mathbb{N} \). *Subformulas* and *subterms* of formulas are inductively defined as follows:

• Every formula is a subformula of itself.
• If \( \varphi \) is a formula, then \( \varphi \) is a subformula of \( \neg \varphi \).
• If \( \varphi \) and \( \psi \) are formulas, then \( \varphi \) and \( \psi \) are subformulas of \( \varphi \land \psi \).

The rule for every other binary connective is analogously defined.
• If \( P \) is a predicate symbol and \( \tau_1, \ldots, \tau_k \) are terms, then \( \tau_1, \ldots, \tau_k \) are subterms of \( P(\tau_1, \ldots, \tau_k) \).
• If \( \varphi \) is a formula and \( \tau \) is a subterm of \( \varphi \), then every subformula (and subterm) of \( \tau \) is a subformula (or subterm) of \( \varphi \).
• If \( \psi \) is a subformula of \( \varphi \), then every subformula (and subterm) of \( \psi \) is also a subformula (or subterm) of \( \varphi \).
The subformula relation is antisymmetric, transitive, and reflexive over formulas; hence it is a partial order over formulas.

To strengthen the inference of the submodel identification process, the associativity and commutativity of connectives and predicates in formulas and functions in terms are accounted for in the subformula and subterm relations. That is, if $\psi_1 \land \psi_2$ is a subformula of a formula, then $\psi_2 \land \psi_1$ is so as well.

### 4.1.2 Abstractions

An abstraction is a way to externalise terms of a formula, and is a formula itself. Abstractions are used to find, from a given submodel, another submodel that is an abstraction of the former and either occurs more frequently throughout the model in question, or is conjectured to be more profitably presolvable, or both. Consider the third constraint of the BHPP model (shown in Figure 2.10) and its subformula

$$P_{in}(x[i] - x[i + 1], \tau),$$

where $\tau$ is the formula

$$f_{++}([f_{+}(f_{+}(13, j), 1) | j], [f_{+}(f_{+}(13, k), 1) | k]).$$

We will refer to the subformula as $\psi$. Notice that if $\psi$ were to be encapsulated within a predicate definition, then $x$ (a 1-dimensional term with domain $\{1, \ldots, 52\}$) and $i$ (a 0-dimensional term with domain $\{1, \ldots, 51\}$) would be the arguments. However, if $x[i]$ and $x[i + 1]$ would be valuated in the call and not in the predicate definition—where two 0-dimensional terms each with domain $\{1, \ldots, 52\}$ were used as arguments, representing $x[i]$ and $x[i + 1]$, instead—then the submodel gets a smaller search space, which may be preferable. This new submodel would be represented by the formula

$$P_{in}(a - b, \tau),$$

where $a$ and $b$ would respectively have the declarations 1..51: a and 1..51: b.

This MZL formula corresponds to the MiniZinc expression

$$(a - b) \text{ in } \{13 \ast j + 1 \mid j \text{ in } -4..3\} \text{ union } \{13 \ast k - 1 \mid k \text{ in } -3..4\}$$

where $a$ and $b$ would respectively have the declarations 1..51: a and 1..51: b.

Note that an abstraction can have a larger search space than the original formula. For example, if we, instead of abstracting both $x[i]$ and $x[i + 1]$, only abstracted one of them, then we would get an abstraction with larger search space than the original formula. This is because we now have $x$, $i$, and (say) $a$ as arguments.

In addition to getting a smaller search space, making a submodel more abstract may allow other parts of the model to use it as well. In some cases, trading a slightly larger search space for more use of the presolved predicate may be beneficial. Suppose that the BHPP model also had an additional fifth constraint that asserted that the topmost card of the black hole was adjacent to the card directly above the Ace of Spades. This new constraint could be expressed as follows:
constraint \((x[2] - x[52]) \text{ in } \{13\cdot j + 1 \mid j \text{ in } -4..3\} \cup \{13\cdot k - 1 \mid k \text{ in } -3..4\}\); 

This constraint is represented by the formula
\[ P_{in}(x[2] - x[52], \tau). \]

Notice the similarity between this formula and \(\psi\). If the aforementioned abstraction of \(\psi\) is encapsulated, then this new formula could also use that predicate by using \(x[2]\) instead of \(x[i]\) as the first argument and \(x[52]\) instead of \(x[i + 1]\) as the second argument.

Abstractions are defined over both terms and formulas, and either is needed for the other.

**Definition 4.3** (Term Abstraction). Let \(D\) be an \(L\)-declaration and let \(k\) range over the non-negative integers \(\mathbb{N}\). The relation of term abstraction is inductively defined as follows:

- Every term is an abstraction of itself.
- If \(\tau\) is a term and \(y\) is an argument variable such that the type \(\tau^D\) is equal to the type \(y^D\), then \(y\) is an abstraction of \(\tau\).
- If \(\tau_1, \ldots, \tau_k\) and \(\tau'_1, \ldots, \tau'_k\) are terms such that, for \(1 \leq i \leq k\), the term \(\tau'_i\) is an abstraction of \(\tau_i\), then, for every term \(\tau\), the term \(\tau[\tau'_1, \ldots, \tau'_k]\) is an abstraction of \(\tau[\tau_1, \ldots, \tau_k]\).
- If \(\tau_1, \ldots, \tau_k\), and \(\tau'_1, \ldots, \tau'_k\) are terms such that, for \(1 \leq i \leq k\), the term \(\tau'_i\) is an abstraction of \(\tau_i\), then, for every function symbol \(f\), the term \(f(\tau'_1, \ldots, \tau'_k)\) is an abstraction of \(f(\tau_1, \ldots, \tau_k)\).
- If \(\tau_1, \ldots, \tau_k\) and \(\tau'_1, \ldots, \tau'_k\) are terms such that, for \(1 \leq i \leq k\), the term \(\tau'_i\) is an abstraction of \(\tau_i\), then the term \([\tau'_1, \ldots, \tau'_k]\) is an abstraction of \([\tau_1, \ldots, \tau_k]\).
- Let \(y_1, \ldots, y_k\) be argument variables. If \(\tau\) and \(\tau'\) are terms such that the term \(\tau'\) is an abstraction of \(\tau\), then the term \([\tau' \mid y_1, \ldots, y_k]\) is an abstraction of \([\tau \mid y_1, \ldots, y_k]\). If \(\varphi\) and \(\varphi'\) are formulas such that \(\varphi'\) is an abstraction of \(\varphi\) (defined for formulas below), then the term \([\varphi' \mid y_1, \ldots, y_k : \varphi]\) is an abstraction of \([\varphi \mid y_1, \ldots, y_k : \varphi]\).
- If \(\varphi\) is a formula and \(\psi\) is an abstraction of \(\varphi\), then \(f_{\text{bool2int}}(\psi)\) is an abstraction of \(f_{\text{bool2int}}(\varphi)\).
- If \(\tau\) is an abstraction of \(\tau'\) and the formula \(\tau'\) is an abstraction of \(\tau''\), then \(\tau\) is an abstraction of \(\tau''\) as well. \(\square\)

The relation of abstraction over terms is antisymmetric, irreflexive, and transitive; it is hence a partial order over terms.

**Definition 4.4** (Formula Abstraction). Let \(D\) be an \(L\)-declaration and let \(k\) range over the non-negative integers \(\mathbb{N}\). The relation of formula abstraction is inductively defined as follows:
• Every formula is an abstraction of itself.

• If \( \tau_1, \ldots, \tau_k \) and \( \tau'_1, \ldots, \tau'_k \) are terms such that, for \( 1 \leq i \leq k \), the term \( \tau'_i \) is an abstraction of \( \tau_i \), then, for every predicate symbol \( P \), the formula \( P(\tau'_1, \ldots, \tau'_k) \) is an abstraction of \( P(\tau_1, \ldots, \tau_k) \).

• If \( \varphi \) is a formula and the formula \( \psi \) is an abstraction of \( \varphi \), then \( \neg \psi \) is an abstraction of \( \neg \varphi \).

• If \( \varphi, \varphi', \psi, \) and \( \psi' \) are formulas such that \( \varphi' \) and \( \psi' \) are abstractions of \( \varphi \) and \( \psi \), respectively, then \( \varphi' \land \psi' \) is an abstraction of \( \varphi \land \psi \).

The rule for every other binary connective is analogously defined.

• If \( \varphi'' \) is an abstraction of \( \varphi' \) and \( \varphi' \) is an abstraction of \( \varphi \), then \( \varphi'' \) is also an abstraction of \( \varphi \).

The relation of abstraction over formulas is antisymmetric, irreflexive, and transitive; hence it is a partial order over formulas.

4.1.3 Generalisations

Generalisation is only used for making the search space of a submodel bigger in order to make it possible for more parts of the model to use the encapsulation. This is done by making the domains of the decision variables and parameters of a submodel larger, or by making a decision variable or parameter that is used at least twice in a submodel two separate decision variables or two separate parameters.

Consider, again, the third constraint of the BHPP model (shown in Figure 2.10) and its subformula

\[ P_{in}(x[i] - x[i + 1], \tau), \]

where \( \tau \) is the formula

\[ f_{++}([f_+(f_+(13, j), 1) | j], [f_+(f_+(13, k), 1) | k]). \]

A generalisation of this subformula is

\[ P_{in}(x[i] - x[i_2 + 1], \tau), \]

where \( i_2 \) is a fresh parameter with the same type as \( i \) but its domain is the whole of \( \mathbb{Z} \). This MZL formula represents the MiniZinc expression

\[ ((x[i1] - x[i2+1]) \text{ in } \{13*j+1 \mid j \text{ in } -4..3\} \text{ union } \{13*k-1 \mid k \text{ in } -3..4\}) \]

where \( i1 \) is a parameter that represents \( i \) and is defined by the parameter variable item 1..51: i1 and where \( i2 \) is a parameter that represents \( i_2 \) and is defined by the parameter variable item int: i2. In this case, the parameter \( i \) has been split into two distinct parameters, where the domain of one of them has been enlarged.

41
The concept of generalisations enables multiple submodels—that look very similar syntactically, but their domains of the decision variables and parameters do not match, or one uses the same decision variable or parameter multiple times where another uses distinct decision variables or parameters; and hence, share a generalisation—to be encapsulated within a predicate definition representing this shared generalisation.

**Definition 4.5** (Term Generalisation). Let $D$ be an $L$-declaration and let $k$ range over the non-negative integers $\mathbb{N}$. The relation of term generalisation is inductively defined as follows:

- Every term is a generalisation of itself.
- If $y$ and $y'$ are argument variables such that the type $y^D$ is a subtype of $y'^D$, then $y$ is a generalisation of $y'$.
- If $\tau_1, \ldots, \tau_k$ and $\tau'_1, \ldots, \tau'_k$ are terms such that, for $1 \leq i \leq k$, the term $\tau'_i$ is a generalisation of $\tau_i$, then, for every term $\tau$, the term $\tau[\tau'_1, \ldots, \tau'_k]$ is a generalisation of $\tau[\tau_1, \ldots, \tau_k]$.
- If $\tau_1, \ldots, \tau_k$ and $\tau'_1, \ldots, \tau'_k$ are terms such that, for $1 \leq i \leq k$, the term $\tau'_i$ is a generalisation of $\tau_i$, then, for every function symbol $f$, the term $f(\tau'_1, \ldots, \tau'_k)$ is a generalisation of $f(\tau_1, \ldots, \tau_k)$.
- If $\tau_1, \ldots, \tau_k$ and $\tau'_1, \ldots, \tau'_k$ are terms such that, for $1 \leq i \leq k$, the term $\tau'_i$ is a generalisation of $\tau_i$, then, for every function symbol $f$, the term $f(\tau'_1, \ldots, \tau'_k)$ is a generalisation of $\tau[\tau_1, \ldots, \tau_k]$.
- Let $y_1, \ldots, y_k$ be argument variables. If $\varphi$ and $\varphi'$ are formulas such that $\varphi'$ is a generalisation of $\varphi$ (defined for formulas below), then the term $[\tau' | y_1, \ldots, y_k]$ is a generalisation of $[\tau | y_1, \ldots, y_k]$.
- If $\varphi$ is a formula and $\psi$ is a generalisation of $\varphi$, then $f_{\text{bool2int}}(\psi)$ is a generalisation of $f_{\text{bool2int}}(\varphi)$.
- If $\tau$ is a generalisation of $\tau'$ and $\tau'$ is a generalisation of $\tau''$, then $\tau$ is also a generalisation of $\tau''$. \qed

The relation of generalisation over terms is antisymmetric, irreflexive, and transitive; hence it is a partial order over terms.

Note that if $\tau$ is a generalisation of $\tau'$, then $\tau$ may contain more free argument variables than $\tau'$, that is, free($\tau$) $\supseteq$ free($\tau'$) with the possibility of strictness.

**Definition 4.6** (Formula Generalisation). Let $D$ be an $L$-declaration and let $k$ range over the non-negative integers $\mathbb{N}$. The relation of formula generalisation is inductively defined as follows:

- Every formula is a generalisation of itself.
• If \(\tau_1, \ldots, \tau_k\) and \(\tau'_1, \ldots, \tau'_k\) are terms such that, for \(1 \leq i \leq k\), the term \(\tau'_i\) is a generalisation of \(\tau_i\), then, for every predicate symbol \(P\), the formula \(P(\tau'_1, \ldots, \tau'_k)\) is a generalisation of \(P(\tau_1, \ldots, \tau_k)\).

• If \(\varphi\) and \(\psi\) are formulas such that \(\psi\) is a generalisation of \(\varphi\), then \(\neg\psi\) is a generalisation of \(\neg\varphi\).

• If \(\varphi, \varphi', \psi,\) and \(\psi'\) are formulas such that \(\varphi'\) and \(\psi'\) are generalisations of \(\varphi\) and \(\psi\), respectively, then \(\varphi' \land \psi'\) is a generalisation of \(\varphi \land \psi\). The rule for every other binary connective is analogously defined.

• If \(\varphi''\) is a generalisation of \(\varphi'\) and \(\varphi'\) is a generalisation of \(\varphi\), then \(\varphi''\) is also a generalisation of \(\varphi\).

The relation of generalisation over formulas is antisymmetric, irreflexive, and transitive; hence it is a partial order over formulas.

Note that abstractions can also split a parameter in the same sense as allowed in generalisations. (However, abstractions cannot enlarge domains.) This means that the concept of generalisations and abstractions overlap. Because this was figured out in the late stages of the thesis work, it is not fixed in the theory due to time reasons.

4.1.4 Submodel Identification Graphs

**Definition 4.7 (SIG).** Given an \(L\)-declaration, an \(L\)-definition, and an \(L\)-constraint set \(\Phi\), a submodel identification graph (SIG) is a 3-tuple \((G, \chi, \lambda)\) where \(G = (V, E)\) is a DAG, \(\chi\) is a labelling function that maps each vertex \(u \in V\) to an \(L\)-formula \(\chi(u)\), and \(\lambda\) is a labelling function that maps each edge in \(E\) to a relation type (denoted \(S\) for subformula, \(A\) for abstraction, and \(G\) for generalisation) such that the following conditions are satisfied:

• For every constraint formula \(\varphi \in \Phi\) and for every subformula \(\psi\) of \(\varphi\), there exists a vertex \(u \in V\) such that \(\chi(u) = \psi\).

• For every constraint formula \(\varphi \in \Phi\) and for every subformula \(\psi\) of \(\varphi\), there exists an edge \((u, w) \in E\) such that \(\lambda(u, w) = S\), where \(\chi(u) = \varphi\) and \(\chi(w) = \psi\).

• Given \(u, w \in V\), then \(\chi(u) = \chi(w)\) if and only if \(u = w\).

• For every edge \((u, w) \in E\) such that \(\lambda(u, w) = A\), then \(\chi(w)\) is an abstraction of \(\chi(u)\).

• For every edge \((u, w) \in E\) such that \(\lambda(u, w) = G\), then \(\chi(w)\) is a generalisation of \(\chi(u)\) and there exists another edge \((u', w) \in E\) (i.e., \(u \neq u'\)) with any \(\lambda(u', w)\).

Notice that a SIG may not have any information about the submodels attained via abstractions or generalisations. In order to specify how much a SIG captures its model, a concept called breadth is used.
Definition 4.8 ($k$-Breadth SIG). A SIG \((G, \chi, \lambda)\) where \(G = (V, E)\) is said to have breadth \(k\), where \(k \in \mathbb{N}\), if the following conditions are fulfilled:

- For every vertex \(u \in V\) let the formula \(\chi(u)\) have \(a\) abstractions, there are \(n\) (where \(n = \min(a, k)\)) distinct vertices \(w_1, \ldots, w_n \in V\) such that, for \(1 \leq i \leq n\), there is an edge \((u, w_i) \in E\) such that \(\lambda(u, w_i) = A\).
- The condition for generalisations is analogously defined. \(\square\)

Note that if a \(k\)-breadth SIG is equal to a \((k + 1)\)-breadth SIG but not equal to a \((k - 1)\)-breadth SIG, for some \(L\)-model, then we call the \(k\)-breadth SIG the full-breadth SIG for that \(L\)-model, and there is exactly one such SIG for every \(L\)-model. In this case, the full-breadth SIG may be referred to as a \(k'\)-breadth SIG, for every \(k' \geq k\).

We use \(k\)-breadth SIGs in the implementation with \(k\) as a command-line parameter, where using a larger \(k\) may result in better submodel encapsulations but will increase the runtime.

4.2 Predicate Encapsulation of Submodels

Consider, once again, the third constraint of the BHPP model (shown in Figure 2.10), which is represented in the MZL language \(L\) by the formula

\[
P(\forall (\text{bool2int}(P\text{in}(\langle x[i] - x[i + 1], \tau\rangle) \mid i)),
\]

where \(\tau\) is the formula

\[
f_+ (\langle f_+ (13, j), 1 \mid j \rangle, [f_+ (f_+ (13, k), 1) \mid k])
\]

And recall the submodel (which we will refer to as \(\varphi\))

\[
P_{\text{in}}(a - b, \tau),
\]

which is an abstraction of the submodel

\[
P_{\text{in}}(x[i] - x[i + 1], \tau),
\]

and where \(a\) and \(b\) are 0-dimensional decision variables with domain \(\{1, \ldots, 52\}\). Let \((G, \chi, \lambda)\), where \(G = (V, E)\), be a SIG of the BHPP model, and let \(v \in V\) be a vertex such that \(\chi(v) = \varphi\).

Suppose that we want to encapsulate \(\varphi\) within a predicate definition. (Note that \(\varphi\) already is a predicate call to \(P_{\text{in}}\); however, since we cannot presolve base language predicates, which \(P_{\text{in}}\) is, the encapsulation is necessary.) In order to do that, a new language \(L'\) is needed that extends the language \(L\) with a fresh predicate symbol, say \(P\). The predicate symbol \(P\) will then be defined to represent \(\varphi\) and be used in place of \(\varphi\).

Since a predicate definition may not have any parameters or decision variables, only argument variables, we need a way to transform a formula to fit this requirement. This
transformation process is called argumentification. The argumentification of \( \varphi \) is another formula

\[
P_{\text{in}}(y_a - y_b, \tau),
\]

where \( y_a \) is a fresh argument variable that represents \( a \) and \( y_b \) is a fresh argument variable that represents \( b \). Call this argumentified formula \( \psi \).

**Definition 4.9** (Argumentification). Given a term \( \tau \) (or formula \( \varphi \)), the argumentification of \( \tau \) (or \( \varphi \)) is the process of replacing all occurrences of every parameter in \( \tau \) (or \( \varphi \)) with a fresh \text{par} argument variable (of the same type as the parameter) and all occurrences of every decision variable in \( \tau \) (or \( \varphi \)) with a fresh \text{var} argument variable (of the same type as the decision variable). □

We can put \( \psi \) into a predicate definition represented by the predicate symbol \( P \). We need to make sure that the type signature is correct for \( P \). Let \( D \) be the \( L \)-declaration the \( L \)-model considered. The type signature for \( P \) should be \( y_a^D \times y_b^D \) and its arguments should be \( y_a \) and \( y_b \), under \( D \). Let \( E \) be the \( L \)-definition the \( L \)-model considered. The definition of \( P \), under \( E \), should be \( \psi \). This process is called *setting the predicate*.

**Definition 4.10** (Setting The Predicate). Let \( D \) be an \( L \)-declaration and \( E \) an \( L \)-definition. Let \( \varphi \) be an \( L \)-formula, without any parameters or decision variables, and let \( P \) be a predicate symbol of \( L' \), where \( L' \) is the extension of \( L \) to also include the predicate symbol \( P \). Suppose that \( \text{free}(\varphi) = \{y_1, \ldots, y_n\} \). Setting the predicate \( P \) to \( \varphi \) is the action of extending \( D \) and \( E \) to \( D' \) and \( E' \) such that the following two conditions are fulfilled:

- \( D' \) is the extension of \( D \) to the language \( L' \) such that \( P^{D'} \) has arity \( n \), the type signature \( y_1^D \times \cdots \times y_n^D \), and the arguments \( y_1, \ldots, y_n \).
- \( E' \) is the extension of \( E \) to the language \( L' \) such that \( P^{E'} \) is defined as the formula \( \varphi \). □

The final thing left in the encapsulation process is to substitute every submodel \( \psi \) in the constraints of the model with an appropriate call to the predicate \( P \). To do this, the SIG of the model, in this case \( \langle G, \chi, \lambda \rangle \), is used to determine what terms and formulas are to be substituted, and with what to substitute each. First, the formula \( \varphi \), represented by the vertex \( v \) in the SIG, is substituted with a predicate call to \( P \). We obtain the predicate call and formula (call it \( \varphi' \))

\[
P(a, b, \tau).
\]

Now in the SIG, we can make the vertex \( v \) represent \( \varphi' \), and this change will then propagate throughout the SIG, as described next.

**Definition 4.11** (Predicate Call Substitution). Let \( L \) be a language and \( L' \) the extension of \( L \) to include also the predicate symbol \( P \). Let \( \varphi \) be an \( L \)-formula and let setting the predicate \( P \) to the argumentified \( \varphi \) result in the \( L' \)-declaration \( D \) and the \( L' \)-definition \( E \).
Let $a_1, \ldots, a_n$ (where $n \in \mathbb{N}$) be the list of the parameters and decision variables present in $\varphi$ and let the argumentification of $\varphi$ result in that, for $1 \leq i \leq n$, the parameter or decision variable $a_i$ is represented by the fresh argument variable $y_i$. The predicate call substitution of $\varphi$ is the formula $P(a_1, \ldots, a_n)$.

**Algorithm 4.1** (Substitution Propagation). Let $\langle G, \chi, \lambda \rangle$, where $G = (V, E)$, be the SIG of an $L$-model. Suppose that $w \in V$ is a vertex. Let $\psi_0$ be the predicate call substitution of $\chi(w)$. The propagation of this substitution, throughout the SIG, results in a new SIG $\langle G, \chi', \lambda \rangle$ that is inductively defined as follows:

1. We have that $\chi'(w) = \psi_0$ holds.

2. For every edge $(u, v) \in E$ where $\chi'(v)$ is defined but $\chi'(u)$ is not, suppose that $\chi(v) = \varphi$ and $\chi'(v) = \psi$ (i.e., $\varphi$ is substituted with $\psi$); the function $\chi'(u)$ is defined as follows:
   - If $\lambda(u, v) = S$, then $\chi'(u)$ is the formula $\chi(u)$ where every subformula $\varphi$ of $\chi(u)$ is substituted with the formula $\psi$.
   - If $\lambda(u, v) = A$, then let $A$ be the set of pairs where every pair $(\tau, a) \in A$ denotes that at least one occurrence of $\tau$ in $\chi(u)$ was abstracted to $a$ in order to obtain the abstraction $\varphi$. Furthermore, we define $\chi'(u)$ to be the formula $\psi$ where, for every pair $(\tau, a) \in A$, every occurrence of $a$ in $\psi$ is substituted with $\tau$.

   This step is repeated until, for every $v \in V$ that $w$ is reachable from in $G$, the function $\chi'$ is defined for $v$.

3. For every vertex $v \in V$ where $\chi'(v)$ is still undefined, set $\chi'(v) = \chi(v)$.

Note that, for generalisations, substitution propagation is not fully figured out, and hence not defined formally here, for time reasons. This might nevertheless not a big problem, since we believe that generalisations are only useful in a very few cases.

### 4.3 Submodel Ranking and Heuristics

We have defined several heuristics that can be used for evaluating the profitability of presolving a vertex in a SIG. There are seven heuristics presented in this thesis. Five heuristics are inspired by the observations of Dekker et al. [3]: argument modesty, argument significance, objective function significance, local variable significance, and solution modesty. The other two heuristics include encapsulation significance and a combined heuristic that uses a combination of the other six heuristics.

Consider, again, the third constraint of the BHPP model (shown in Figure 2.10), which is represented in the MZL language $L$ by the formula

$$P_{y}([f_{bool2int}(P_{in}(x[i] - x[i + 1], \tau)) | i])$$

46
where $\tau$ is the formula

$$f_{++}(\lfloor f_{+}(13, j), 1 \rfloor, j, [f_{+}(13, k), 1 \rfloor, k]).$$

Also recall the submodel (which we will refer to as $\psi$)

$$P_{in}(a - b, \tau),$$

which is an abstraction of the submodel (which we will refer to as $\varphi$)

$$P_{in}(x[i] - x[i + 1], \tau),$$

and where $a$ and $b$ are 0-dimensional decision variables with domain $\{1, \ldots, 52\}$. Let $\langle G, \chi, \lambda \rangle$, where $G = (V, E)$, be a SIG of the BHPP model, and let $v \in V$ be the vertex such that $\chi(v) = \varphi$. We will explore the heuristics using $\varphi$.

Every heuristic takes a vertex and gives it a score, which represents the profitability of presolving the submodel represented by that vertex.

Since many formulas depend on a parameter, the score has a lower and upper bound; the lower bound gives the best possible score (the parameters have an optimal assignment), and the upper bound the worst possible score (the parameters have a pessimal assignment).

**Definition 4.12 (Scores).** A score is a pair of numbers $s = (\hat{s}, \tilde{s})$ where $\hat{s}$ is the lower bound and $\tilde{s}$ the upper bound. For these pairs we define the operations $+, - , \cdot, \div,$ and exponentiation as follows:

$$\langle \hat{s}_1, \tilde{s}_1 \rangle + \langle \hat{s}_2, \tilde{s}_2 \rangle = \langle \hat{s}_1 + \hat{s}_2, \tilde{s}_1 + \tilde{s}_2 \rangle$$

$$a + \langle \hat{s}, \tilde{s} \rangle = \langle a + \hat{s}, a + \tilde{s} \rangle = \langle \hat{s}, \tilde{s} \rangle + a$$

$$\langle \hat{s}_1, \tilde{s}_1 \rangle - \langle \hat{s}_2, \tilde{s}_2 \rangle = \langle \hat{s}_1 - \hat{s}_2, \tilde{s}_1 - \tilde{s}_2 \rangle$$

$$a - \langle \hat{s}, \tilde{s} \rangle = \langle a - \hat{s}, a - \tilde{s} \rangle$$

$$\langle \hat{s}, \tilde{s} \rangle - a = \langle \hat{s} - a, \tilde{s} - a \rangle$$

$$\langle \hat{s}_1, \tilde{s}_1 \rangle \cdot \langle \hat{s}_2, \tilde{s}_2 \rangle = \langle \hat{s}_1 \cdot \hat{s}_2, \tilde{s}_1 \cdot \tilde{s}_2 \rangle$$

$$a \cdot \langle \hat{s}, \tilde{s} \rangle = \langle a \cdot \hat{s}, a \cdot \tilde{s} \rangle = \langle \hat{s}, \tilde{s} \rangle \cdot a$$

$$\langle \hat{s}_1, \tilde{s}_1 \rangle \div \langle \hat{s}_2, \tilde{s}_2 \rangle = \langle \hat{s}_1 \div \hat{s}_2, \tilde{s}_1 \div \tilde{s}_2 \rangle$$

$$a \div \langle \hat{s}, \tilde{s} \rangle = \langle a \div \hat{s}, a \div \tilde{s} \rangle$$

$$\langle \hat{s}, \tilde{s} \rangle \div a = \langle \hat{s} \div a, \tilde{s} \div a \rangle$$

$$\langle \hat{s}_1, \tilde{s}_1 \rangle^{\hat{s}_2, \tilde{s}_2} = \langle (\hat{s}_1)^{\hat{s}_2}, (\tilde{s}_1)^{\tilde{s}_2} \rangle$$

$$a^{\hat{s}, \tilde{s}} = \langle a^{\hat{s}}, a^{\tilde{s}} \rangle$$

$$\langle \hat{s}, \tilde{s} \rangle^a = \langle (\hat{s})^a, (\tilde{s})^a \rangle \quad \square$$
For example, given the scores \( \langle 1, 2 \rangle \) and \( \langle 3, 4 \rangle \), we can perform subtraction (e.g., \( \langle 1, 2 \rangle - \langle 3, 4 \rangle = \langle -2, -2 \rangle \)) and exponentiation (e.g., \( \langle 1, 2 \rangle^{(3,4)} = \langle 1^3, 2^4 \rangle = \langle 1, 16 \rangle \)).

Since scores consist of upper and lower bounds, we could consider ways of comparing the scores. These ways of comparing scores are called **attitudes**, and we explain three attitudes: optimistic, pessimistic, and balanced.

**Definition 4.13 (Attitudes of Comparing Scores).** Let \( s_1 = \langle \hat{s}_1, \check{s}_1 \rangle \) and \( s_2 = \langle \hat{s}_2, \check{s}_2 \rangle \) be two scores. The optimistic, pessimistic, and balanced attitudes are defined as follows:

- **Optimistic:** \( s_1 \leq s_2 \iff \hat{s}_1 \leq \hat{s}_2 \)
- **Pessimistic:** \( s_1 \leq s_2 \iff \check{s}_1 \leq \check{s}_2 \)
- **Balanced:** \( s_1 \leq s_2 \iff \frac{\hat{s}_1 + \check{s}_1}{2} \leq \frac{\hat{s}_2 + \check{s}_2}{2} \)

**Definition 4.14 (Lower and Upper Bounds of Terms).** Let \( D \) be an L-declaration and \( E \) an L-definition. The lower and upper bounds of the domain of a term \( \tau \) are denoted \( \text{bounds}(\tau) \), a score, and are defined as follows:

\[
\text{bounds}(\tau) = \langle \min \text{dom}(\tau^E, D), \max \text{dom}(\tau^E, D) \rangle
\]

First, we will explain and define the argument modesty heuristic, which uses the argument modesty metric, where the lower the score, the better. This heuristic basically says that the fewer arguments the argumentified formula (submodel) has, the better. For example, the argument modesty metric value of \( \psi \) is \( \langle 2 \cdot 52, 2 \cdot 52 \rangle \) since \( a \) and \( b \) have a domain of size 52 regardless of the instance.

**Definition 4.15 (Argument Modesty).** The **argument modesty metric** value \( h \) of an argumentified formula \( \varphi \) is defined as follows:

\[
h(\varphi) = \sum_{y \in \text{free}(\psi)} \left( \prod_{\tau \in \text{dim}(y)} \text{bounds}(\tau) \right).
\]

The **argument modesty heuristic** is to pick a submodel with the lowest argument modesty metric value, with tie-breaking on picking the first found such submodel.

Secondly, we have the argument significance heuristic, which uses the argument significance metric, which is a measure of how many times the same variable is used beyond the first occurrence! Here, a higher score is better. The argument significance metric value of \( \psi \) is 0 since every argument variable is used exactly once.

**Definition 4.16 (Argument Significance).** The **argument significance metric** function \( h \) is defined for formulas, term-argument-variable pairs, and formula-argument-variable...
pairs as follows:

\[
    h(\varphi) = 1 + \sum_{y \in \text{free}(\varphi)} (h(\varphi, y) - 1)
\]

\[
    h(\varphi, y) = \begin{cases} 
        \langle 1, 1 \rangle & \text{if } \tau = y. \\
        \langle 0, 0 \rangle & \text{if } \tau \text{ is a constant, parameter, decision variable, or argument variable } \neq y. \\
        h(\varphi, y) + \sum_{i=1}^{k} h(\tau_i, y) & \text{if } \tau = [\tau_1, \ldots, \tau_k]. \\
        h(\tau', y) + \sum_{i=1}^{k} h(\tau_i, y) & \text{if } \tau = \tau'[\tau_1', \ldots, \tau_k'] \\
        h(\varphi, y) + \sum_{i=1}^{k} h(\tau_i, y) \cdot \text{bounds}(y_i) & \text{if } \tau = \tau'[\psi_1, \ldots, \psi_k : \varphi]. \\
        h(\varphi, y) \cdot \text{bounds}(y) & \text{if } \tau = f_{\text{bool2int}}(\varphi).
    \end{cases}
\]

The argument significance heuristic is to pick a submodel with the highest argument significance metric value, with tie-breaking on picking the first found such submodel.

Thirdly, we define the objective function significance heuristic, which uses the objective function significance metric, where a higher score is better. The objective function significance metric value is 0 if no decision variables are shared between the objective term and the formula in question, 1 if they share exactly decision variable, etc. For example, the objective function significance of \( \psi \) (page 47) is 0 since there is no objective function.

**Definition 4.17** (Objective Function Significance). Let \( \langle G, \chi, \lambda \rangle \), where \( G = (V, E) \), be a SIG of a model and let \( v \in V \) be a vertex such that \( \chi(v) = \varphi \). Let \( \langle G, \chi', \lambda \rangle \) be the SIG resulting from substitution propagation from the predicate call substitution of \( \varphi \) into predicate \( P_0 \). Let \( D \) be the \( L \)-declaration and \( E \) the \( L \)-definition of the model, where \( \tau_0 \) is the objective term that has the decision variables \( D = \{x_1, \ldots, x_n\} \). Let \( \{\psi'_1, \ldots, \psi'_k\} \) be the model constraints resulting from the substitution propagation, that is, for \( 1 \leq i \leq k \), we have \( \chi'(w_i) = \psi'_i \). The objective function significance metric value
$h$ is then calculated, using a help function $g$, as follows:

\[
h(\varphi) = \left| \left[ D \cap \bigcup_{\psi \in \{\psi_1', \ldots, \psi_k'\}} g(\psi) \right] \right|
\]

\[
g(\tau) = \begin{cases} 
\emptyset & \text{if } \tau \text{ is a constant, parameter, decision variable, or argument variable.} \\
 g(\tau') \cup \bigcup_{i=1}^{k} g(\tau_i) & \text{if } \tau = \tau'[\tau_1, \ldots, \tau_k]. \\
 \bigcup_{i=1}^{k} g(\tau_i) & \text{if } \tau = [\tau_1, \ldots, \tau_k], \text{ or } \tau = f(\tau_1, \ldots, \tau_k). \\
g(\tau') & \text{if } \tau = [\tau' | y], \text{ or } \tau = [\tau' | y_1, \ldots, y_k]. \\
g(\tau') \cup g(\varphi) & \text{if } \tau = [\tau' | y : \varphi], \text{ or } \tau = [\tau' | y_1, \ldots, y_k : \varphi]. \\
g(\varphi) & \text{if } \tau = f_{\text{bool2int}}(\varphi). \\
g(\tau') & \text{if } \tau = y_1, \ldots, y_k : \tau'. 
\end{cases}
\]

\[
g(\varphi) = \begin{cases} 
\emptyset & \text{if } \varphi = \bot \text{ or } \varphi = \top. \\
 \bigcup_{i=1}^{k} g(\tau_i) & \text{if } \varphi = P(\tau_1, \ldots, \tau_k) \text{ and } P \neq P_0. \\
 \bigcup_{i=1}^{k} g(\tau_i) \cup S_i & \text{if } \varphi = P_0(\tau_1, \ldots, \tau_k), \text{ where } S_i \text{ denotes the decision variables of } \tau_i, \text{ for } 1 \leq i \leq k. \\
g(\psi) & \text{if } \varphi = \neg \psi. \\
g(\psi_1) \cup g(\psi_2) & \text{if } \varphi = \psi_1 \oplus \psi_2, \text{ where } \oplus \text{ is a binary connective.}
\end{cases}
\]

The objective function significance heuristic is to pick a submodel with the highest objective function significance metric value, with tie-breaking on picking the first found such submodel.

The local variable heuristics uses the local variable metric, which counts how many local decision variables are introduced in the predicate. Here, a higher score denotes a higher profitability of presolving. For example, the local variable significance metric value of $\psi$ is 0 since it does not contain any let expressions, and hence no local variables.

**Definition 4.18 (Local Variable Significance).** The local variable significance metric
function $h$, of a term $\tau$ and a formula $\varphi$, is defined as follows:

$$h(\tau) = \begin{cases} 
(0,0) & \text{if } \tau \text{ is a constant, parameter, decision variable, or argument variable.} \\
h(\tau') + \sum_{i=1}^{k} h(\tau_i) & \text{if } \tau = [\tau_1, \ldots, \tau_k]. \\
h(\tau') & \text{if } \tau = [\tau' | y], \text{ or } \tau = [\tau' | y_1, \ldots, y_k]. \\
h(\tau') + h(\varphi) & \text{if } \tau = [\tau' | y : \varphi], \text{ or } \tau = [\tau' | y_1, \ldots, y_k : \varphi]. \\
h(\varphi) & \text{if } \tau = f_{\text{bool2int}}(\varphi). \\
(1,1) + h(\tau') & \text{if } \tau = y_1, \ldots, y_k : \varphi'. 
\end{cases}$$

$$h(\varphi) = \begin{cases} 
(0,0) & \text{if } \varphi = \bot \text{ or } \varphi = \top. \\
\sum_{i=1}^{k} h(\tau_i) & \text{if } \varphi = P(\tau_1, \ldots, \tau_k). \\
h(\psi) & \text{if } \varphi = \neg \psi. \\
h(\psi_1) + h(\psi_2) & \text{if } \varphi = \psi_1 \oplus \psi_2, \text{ where } \oplus \text{ is a binary connective.}
\end{cases}$$

The local variable significance heuristic is to pick a submodel with the highest local variable significance metric value, with tie-breaking on picking the first found such submodel.

The search space modesty heuristic uses an approximation of how many solution the predicate has: the search space modesty metric. Since it is hard to calculate the solution space, we approximate the search space (which is easier) and use that instead. A lower score is better in this case. For example, the search space modesty metric value of $\psi$ is $\langle 52 \cdot 52, 52 \cdot 52 \rangle$ since the search space consists of $a$ and $b$ and both their domains are $\{1, \ldots, 52\}$, which yields that number of combinations of assignments.

**Definition 4.19 (Search Space Modesty).** The solution modesty metric function $h$ of a formula $\varphi$ is defined as follows:

$$h(\varphi) = \prod_{x \in \text{free}(\varphi)} \text{bounds}(y)^{|\text{dim}(y)|}$$

The search space modesty heuristic is to pick a submodel with the lowest search space modesty metric value, with tie-breaking on picking the first found such submodel.

The encapsulation significance heuristic uses the encapsulation significance metric, which, given a submodel, approximates how many times (including inside array comprehensions, where the number of times is relative to the size of the array) the predicate call to the encapsulated submodel will be in the final model once the submodel is encapsulated. A higher score is better in this case.

**Definition 4.20 (Encapsulation Significance).** Let $\langle G, \chi, \lambda \rangle$, where $G = (V,E)$, be a SIG of a model and let $v \in V$ be a vertex such that $\chi(v) = \varphi$. Let $\langle G, \chi', \lambda \rangle$ be the SIG resulting from substitution propagation from the predicate call substitution of $\varphi$
into predicate $P_0$. Let $\mathcal{D}$ be the $L$-declaration and $\mathcal{E}$ the $L$-definition of the model. Let $\{\psi'_1, \ldots, \psi'_k\}$ be the model constraints resulting from the substitution propagation, that is, for $1 \leq i \leq k$, we have $\chi'(w_i) = \psi'_i$. The encapsulation significance metric function $h$ of the given submodel $\varphi$ is defined, using the help function $g$, as follows:

$$h(\varphi) = \sum_{\psi \in \{\psi'_1, \ldots, \psi'_k\}} g(\psi)$$

$$g(\tau) =
\begin{cases}
(0, 0) & \text{if } \tau \text{ is a constant, parameter, decision variable, or argument variable.} \\
g(\tau') + \sum_{i=1}^k g(\tau_i) & \text{if } \tau = \tau'[\tau_1, \ldots, \tau_k]. \\
\sum_{i=1}^k g(\tau_i) & \text{if } \tau = [\tau_1, \ldots, \tau_k], \text{ or } \tau = f(\tau_1, \ldots, \tau_k). \\
g(\tau') & \text{if } \tau = [\tau'| y], \text{ or } \tau = [\tau'| y_1, \ldots, y_k]. \\
g(\tau') + g(\varphi) & \text{if } \tau = [\tau'| y : \varphi], \text{ or } \tau = [\tau'| y_1, \ldots, y_k : \varphi]. \\
g(\varphi) & \text{if } \tau = f_{\text{bool2int}}(\varphi). \\
g(\tau') & \text{if } \tau = y_1, \ldots, y_k : \tau'.
\end{cases}$$

$$g(\varphi) =
\begin{cases}
(0, 0) & \text{if } \varphi = \bot \text{ or } \varphi = \top. \\
\sum_{i=1}^k g(\tau_i) & \text{if } \varphi = P(\tau_1, \ldots, \tau_k) \text{ and } P \neq P_0. \\
(1, 1) + \sum_{i=1}^k g(\tau_i) & \text{if } \varphi = P_0(\tau_1, \ldots, \tau_k). \\
g(\psi) & \text{if } \varphi = \neg \psi. \\
g(\psi_1) + g(\psi_2) & \text{if } \varphi = \psi_1 \oplus \psi_2, \text{ where } \oplus \text{ is a binary connective.}
\end{cases}$$

The encapsulation significance heuristic is to pick a submodel with the highest encapsulation significance metric value, with tie-breaking on picking the first found such submodel.

In order to facilitate the usability of all the heuristics together, a heuristic combining them, called the mix heuristic, has been created. The scores are normalised to a 0–1-scale and some scales are flipped such that higher values are better.

**Definition 4.21 (Mix).** The mix heuristic is to pick a submodel with the highest sum every metric value of the submodel normalised to a 0–1-scale and, if necessary, flipped such that a higher score is better instead of a lower one. The tie-breaking is on picking the first found such submodel.

5 Implementation and Application

We propose, design, and explain the implementation of a tool (referred to as Mzn-Encaps) that follows the concepts and theory described in Sections 3 and 4.

The tool Mzn-Encaps is implemented in Haskell; installation instructions can be found in Appendix B.

We explain the central features and functionality of the tool in Section 5.1. In Section 5.2, an overview of the design of Mzn-Encaps is shown. The representation of
the data structures is explained in Section 5.3. In Section 5.4 the lexing and parsing of MiniZinc models are presented, and in Section 5.5 we show how the theory and concepts are applied in Mzn-Encaps.

5.1 Features and Functionality

Mzn-Encaps has some key features that make it useful. Mzn-Encaps is called from a command line interface via 'mzn-encaps m' where m is the path to a MiniZinc model.

There are some call parameters that can be used to determine the behaviour of Mzn-Encaps; all of these (except the ones used for debugging) are listed below:

**Output Format.** The output is either a new MiniZinc model where the encapsulation is performed (selected via the flag '--format=model', and is the default output format), or the SIG that has been generated while performing submodel identification of the input MiniZinc model (selected via the flag '--format=graph').

**Identification Heuristic Parameter.** This parameter dictates how the encapsulations will be evaluated and ranked when the SIG is constructed. This parameter is set by the flag '--heur1=n' where n is the heuristic to use (the default is n = 1). The heuristics are chosen from the list in Section 4.3 with the number corresponding to the order of introduction (i.e., n = 1 corresponds to argument modesty, since it is introduced first, and n = 2 corresponds argument significance since it is introduced second, etc.), with the addition of n = 0, which means always picking by order of occurrence of the submodels encountered.

**Encapsulation Heuristic Parameter.** This parameter is similar to the identification heuristic parameter; however, this one specifies how the submodels will be evaluated and ranked when selecting candidates for encapsulation. This parameter is set by the flag '--heur2=n' where n is the heuristic to use (with the same convention of numbers corresponding to heuristics as for the identification heuristic parameter). The default is the same value as selected for the identification heuristic parameter.

**Attitude Parameter.** This parameter determines the attitude used both when creating the SIG and when selecting the encapsulation, and is set via the flag '--att=r' where the string r is either 'optimistic', 'pessimistic', or 'balanced' (the default is r = balanced).

**Rank Parameter.** With this parameter, the modeller can choose specifically which encapsulation to perform. This parameter is set via the flag '--rank=r' where the positive integer r denotes the rth best encapsulation according to the chosen heuristics (the default is r = 1).
Breadth Parameter. This parameter is set via the flag `--breadth=b`, where the positive integer $b$ denotes the breadth of the SIG generation process (the default is $b=3$)

5.2 Overview of the Design

In Figure 5.1, the data flow of Mzn-Encaps can be seen. A MiniZinc model is fed into the Translator (explained in Section 5.4), which interprets and parses the MiniZinc syntax of the model and produces a concrete syntax tree (CST). The CST is then fed into the Identifier (explained in Section 5.5.1), which analyses the CST and, with the help of the Ranker (explained in Section 5.5.2), produces a $k$-breadth SIG (the representation of which is explained in Section 5.3) where $k$ is the value of the breadth parameter set by the modeller. Then, this SIG is fed into the Encapsulator (explained in Section 5.5.3), which, via the Ranker again, finds a best submodel (according to the selected encapsulation heuristic) to encapsulate. The Encapsulator creates a new predicate definition that represents this submodel and, in the SIG, substitutes the submodel with a predicate call to this predicate. This substitution is propagated throughout the SIG, creating a new SIG. This SIG is then passed to the Converter, which translates the SIG back into MiniZinc syntax, creating a MiniZinc model, which is output to standard output.

5.3 Representation

It is important to choose a good representation of the data structures that relate to the concepts of the theory since the analysis algorithms and the other parts using the data structures should not be unnecessarily inefficient. (We allow them to be somewhat subefficient for time reasons.

In Mzn-Encaps, the SIG representation contains several things. Apart from the DAG itself—with the vertices and the subformulas they represent, and the edges connecting the vertices—the SIG includes an environment, which contains information on the parameters, decision variables, predicate definitions, and function definitions of the MiniZinc model. Each of these is stored and associated with a type or type signature, and will be used to do simple type checking and when encapsulating a submodel.

A DAG is represented as a list of vertices (each represented as a MiniZinc expression) and a list of directed edges (each represented as a pair of integers $i$ and $j$, where the edge goes from the $i$th vertex to the $j$th vertex).

The score, which each heuristic returns given a submodel, is represented as a pair of arbitrarily big integers. When normalising, instead of normalising to 0–1-scale (using floats or similar), we normalise to a $0–10^{15}$-scale (using integers). This is so that the same score data structure can be used throughout the program.

5.4 Translator: Syntactic and Semantic Analysis

The translator module is responsible for converting a MiniZinc model from text format into a concrete syntax tree (CST). It performs syntactic and (to some degree) semantic
analyses of MiniZinc models.

The translator consists of two parts: a lexer and a parser. A data flow diagram of the translator module can be seen in Figure 5.1.

Note that the Translator does not perform thorough type checking; thus, a MiniZinc textfile needs to compile using a MiniZinc compiler, such as minizinc, or similar, first.

The lexer takes as input a MiniZinc model (which is in text format) and turns it into a list of tokens. The tokens are identifiers, string literals, integer literals, float literals, and symbols and keywords defined in the MiniZinc 2.1.0 specification [12]. We use a lexer since it will make the parsing easier.

The parser takes a list of tokens and turns it into a CST. A CST is structured in alignment to the rules of the MiniZinc grammar explained in the MiniZinc 2.1.0 specification [12]. However, some modifications have been done: one to handle the rules of associativity and priority of operations directly in the CST, others are because the specification seems to be incomplete in some cases.

The parser is written with the use of the monadic parser combinator library Parsec[^6].

[^6]: More information can be found at [https://wiki.haskell.org/Parsec](https://wiki.haskell.org/Parsec)
5.5 Application of Theory: Identifying, Ranking, and Encapsulating

The application of our theory (Section 4) is in some cases straightforward and need not to be explained. However, some hacks have been performed because of the nature of either Haskell or the computationally inconvenient mathematics of the theory. We do not explain all the hacks, only the most significant ones, for time reasons; the interested reader is referred to the source code.

There are three modules: the Identifier (Section 5.5.1), which is the application of the submodel identification concepts; the Ranker (Section 5.5.2), which is the application of the submodel ranking heuristics and scoring; and the Encapsulator (Section 5.5.3), which is the application of the submodel encapsulation concepts.

5.5.1 Submodel Identifier

Before we describe how the submodel identification process is implemented, an additional concept needs to be established. This concept is called immediate subformulas, which are defined for terms and formulas as follows:

**Definition 5.1** (Immediate Subformula for Terms). Let $k$ range over the non-negative integers $\mathbb{N}$. The *immediate subformulas* of terms are inductively defined as follows:

- Every constant, parameter, decision variable, and argument variable has no immediate subformulas.
- If $\tau_1, \ldots, \tau_k$ are terms, then, for every term $\tau$, the immediate subformulas of $\tau_1, \ldots, \tau_k$ are immediate subformulas of $\tau[\tau_1, \ldots, \tau_k]$.
- If $\tau_1, \ldots, \tau_k$ are terms, then, for every function symbol $f$, the immediate subformulas of $\tau_1, \ldots, \tau_k$ are immediate subformulas of $f(\tau_1, \ldots, \tau_k)$.
- If $\tau_1, \ldots, \tau_k$ are terms, then the immediate subformulas of $\tau_1, \ldots, \tau_k$ are immediate subformulas of $[\tau_1, \ldots, \tau_k]$.
- If $\tau$ is a term, $y_1, \ldots, y_k$ are argument variables, and $\varphi$ is a formula, then the immediate subformulas of $\tau$ are immediate subformulas of $[\tau | y_1]$, of $[\tau | y_1, \ldots, y_k]$, of $[\tau | y_1 : \varphi]$, and of $[\tau | y_1, \ldots, y_k : \varphi]$ (with $\varphi$ as an immediate subformula of the latter two).
- If $\varphi$ is a formula, then $\varphi$ is an immediate subformula of the term $f_{\text{bool}2\text{int}}(\varphi)$. □
**Definition 5.2** (Immediate Subformulas of Formulas). Let $E$ be an $L$-definition and let $k$ range over the non-negative integers $\mathbb{N}$. The *immediate subformulas* of formulas are inductively defined as follows:

- If $\varphi$ is a formula, then $\varphi$ is an immediate subformula of $\neg \varphi$.

- If $\varphi$ and $\psi$ are formulas, then $\varphi$ and $\psi$ are immediate subformulas of $\varphi \land \psi$.

  The rule for every other binary connective is analogously defined.

- If $P$ is a predicate symbol and $\tau_1, \ldots, \tau_k$ are terms, then the immediate subformulas of $\tau_1, \ldots, \tau_k$ are immediate subformulas of $P(\tau_1, \ldots, \tau_k)$.

  □

Note that the immediate subformula relation is anti-transitive, irreflexive, and asymmetric.

From the CST of a model, the Identifier produces a SIG. The process contains three phases:

1. An initial DAG is generated from the constraints of the model. Here each constraint gets represented by its own vertex in the graph. These initial vertices are special because any alteration to these will result in differently formulated constraints.

2. The DAG is expanded by, for each vertex, finding all the immediate subformulas of the formula of that vertex, representing each found immediate subformula by a new vertex in the graph, and inserting an edge from the vertex to every vertex representing an immediate subformula. This process is repeated until no more vertices can be added, where the resulting DAG is a (0-breadth) SIG of the model. If two subformulas are equal, then they will be merged, that is, represented by the same vertex. We also keep track of every new parameter that has to be introduced, when finding a subformula to an expression with local variables, and what its original parameter name was in the generator. This is so that we can easily propagate changes in the SIG.

3. The SIG is expanded by, for each of the already existing vertices, finding the $k$ best (according to the ranking function used) abstractions. (Generalisations are not yet implemented because of time reasons.) If two abstractions are equal formulas, then they are merged to be represented by the same vertex. Note that this step is only performed once and for the vertices already existing in the SIG when this step is reached. For every new abstracted term, we introduce a new argument variable and keep track of the term that each introduced argument variable has abstracted. Finally, the SIG is completed and is a $k$-breadth SIG.

---

7 Note that the commutativity property of connectives and the built-in functions are taken into account when checking for equality, in order to strengthen the inference of the submodel identification process. However, associativity and other properties are not considered for time reasons.

8 See footnote 7.
Note that not all abstractions, as explained in the theory, are considered here. Abstractions of Boolean expressions and abstractions of a subset of the elements of array and set literals are never considered, for runtime optimisation reasons.

Also note that when we introduce parameters (in subformulas) and argument variables (in abstractions), we have a process to derive the type of that introduced parameter or argument variable. This is done by recursively deriving the type of a MiniZinc expression in a structural manner. For example, if an expression is a function call, then the type of the expression is the return type of the function, except that the class of the expression is the supreme class of the class of the return type of the function and the classes of the arguments of the function call. Another example is when we have an expression that is the union of two subexpressions, then the type of the expression has the supreme class of the two subexpressions and has the union of the domains of the subexpressions as domain. The subexpressions must agree on other criteria of a type, and hence the type of the expression will also agree on these criteria. Furthermore, when calculating the type of an identifier, the implemented bounds function (explained next, in Section 5.5.2) is used.

This process is very crude, due to time reasons, and most likely contains several bugs.

5.5.2 Submodel Ranker

This process is similar to the theory presented in Section 4.3 but there are some technical differences, explained here. There are also some differences in how the heuristics work.

The bounds function, as explained in the theory (Section 4), is used in the implementation. However, because of time reasons, in the implementation the function is very simplified, often giving over-approximations of the actual bounds (however, the actual bounds are always included). The implementation of each of the heuristics explained in the theory is a direct application of the theory but using the simplified bounds function implementation.

The ranking of submodels is a ranking function that sorts the list of the submodels in a SIG. Every submodel is ranked with the user-chosen ranking function, and put in descending order of the list.

Note that the operations for scores are not implemented directly as explained in the theory. This is bug in the code noticed too late and not fixed due to time reasons.

5.5.3 Submodel Encapsulator

The Encapsulator is the part of MZN-ENCAPS that performs the encapsulation. This process is similar to the theory, but there are some differences, which are explained here.

The propagation algorithm uses breadth-first traversal and for each visited vertex replaces every old expression (the original expression that forms the submodel that has been encapsulated) with a new expression using a predicate call to the encapsulated submodel. Given a SIG with an environment, a ranker, and an attitude, the submodel encapsulation process works as follows:
1. The list of all vertices in the SIG is sorted according to the ranker and the attitude, resulting in a list where the first element is the best, and the second element is the second best, etc.

2. Taking the \( r \)th-best vertex \( v \) (where \( r \) is the provided rank of the submodel to encapsulate), a predicate call is generated. This process is a direct application of “Setting the predicate” (Definition 4.10).

3. In the SIG, the MiniZinc expression of the vertex \( v \) (called the old \( v \) expression) is replaced with the generated predicate call (called the new \( v \) expression).

4. A queue is created with all the parent edges, including the relationship (subformula or abstraction), of the vertex \( v \), where each edge in the queue is associated with an old part (the old \( v \) expression) and a new part (the new \( v \) expression). (If the queue is empty, then the process is done.)

5. Take the first element in the queue, which consists of an edge (connecting a parent vertex \( u \) and a child vertex \( w \)), a relationship, an old part, and a new part. One of the two following steps is performed:

   - If the relationship is “subformula”, then every occurrence of the old part in the formula of \( u \) is replaced with the new part. Then, the introduced argument variables (if any) are replaced with the original local variable (if they are present in \( u \)), resulting in a new formula. After that, the queue is extended with all the parent edges (if any), including the relationship, of vertex \( u \), with the formula of \( u \) as the old part and the new formula as the new part. Finally, we replace the formula of \( u \) with the new formula.

   - If the relationship is “abstraction”, then every occurrence of an introduced argument variable in the new part is replaced with the original term that was abstracted, resulting in a new formula. Only this operation needs to be performed since it is guaranteed that an abstraction is a leaf in the SIG, by construction; hence, the new part is always just a predicate call in this case. After this, the queue is extended with all the parent edges (if any), including the relationship, of vertex \( u \), with the formula of \( u \) as the old part and the new formula as the new part. Finally, we replace the formula of \( u \) with the new formula.

6. If the queue is not empty, then go back to step 5 else the process is done.

6 Evaluation

MZN-ENCAPS is evaluated with different configurations of the heuristics and other parameters.

The tool is evaluated using two MiniZinc models: one is the BHPP model used throughout this thesis, the other is a model of the block party metacube problem. In
Table 6.1: Computer specifications

Section 6.1, the setup of the machine that will be used to run all the tests is explained. In Section 6.2, the problem and a model of the block party metacube problem are presented. In Section 6.3, the runtime for Mzn-Encaps is evaluated, and, in Section 6.4, the quality of the models output by Mzn-Encaps is evaluated.

There are two versions of both models: a manually encapsulated version, which will be referred to as the Manual model, and a version where an encapsulated submodel from the Manual model has been decapsulated (resulting in a model with no auto-tabling annotation and the encapsulated submodels put back into their respective constraints), which will be referred to as the Bare model. We do this instead of just removing the auto-tabling annotation because the tool has no means of detecting good, already existing predicate definitions to presolve. Furthermore, Mzn-Encaps outputs automatically encapsulated models, which we will refer to as Automatic models (under some specified heuristics and parameters).

(Note that the search annotations from a Manual model are kept in the corresponding Bare model and in every corresponding Automatic model.)

6.1 Setup

All tests were run on the same machine with the specifications shown in Table 6.1.

When Mzn-Encaps outputs an encapsulated MiniZinc model (search annotations are kept), this model is then run using the Gecode MiniZinc backend extended with the auto-tabling tool developed by Dekker [7, 8].

On all tests we had a 15 minute time out for the encapsulation process, a 15 minute time out for the flattening process, and a 15 minute time out for the solving process.

6.2 The Block Party Metacube Problem

The block party metacube problem (BPMP) consists of eight cubes and 64 icons, where each icon is characterised by three attributes (shape, colour, and pattern) with four possible values for each attribute. Each corner of each side of each cube features one of these icons. A party is made of four cubes placed in a two-by-two square, making the four sides that are face-up form a 16-icon grid, such that the four central icons of the up-facing grid have, for each of the three attributes, either distinct or equal values.
For example, the four icons can have the same shape, the same pattern, but all different colours. A *block party metacube* is a placement of the eight cubes forming a two-by-two-by-two metacube such that each of the six sides of the metacube forms a party.

The MiniZinc model used in the evaluation used was designed by Jean-Noël Monette with a manually performed predicate encapsulation by Dekker.

6.3 Runtime Performance of the Encapsulation Process

In Table 6.2, the runtime of the encapsulation process of the two Bare models for most of the heuristics can be seen. The objective function significance heuristic is not used since they are satisfaction problems. Note that the identification and encapsulation heuristic parameters were the same for all the tests. A breadth of $k = 100$ and the balanced attitude were used. As can be seen from the table, the encapsulation times are manageable, and remember that the encapsulation process is only run once, and not per instance or otherwise.

Notice that increasing the breadth increases the runtime. This has to do with the fact that the higher the breadth, the bigger the SIG (up to a point), which results in more submodels to consider both when creating the SIG and when finding a best submodel.

6.4 Comparison of Manually and Automatically Encapsulated Models

For both the BHPP (Section 6.4.1) and the BPMP (Section 6.4.2), the Automatic models are compared with the Manual and Bare models. Both the runtimes (combined solving

---

9 This problem is based on a puzzle game, called ‘Block Party’, introduced by Three Pixel Heart LLC. More information about the puzzle game can be found on their website [http://w.3pxh.com/](http://w.3pxh.com/).

10 At [https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models](https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models), the model, with the manual encapsulation, can be found.
Table 6.2: Runtime of MZN-ENCAPS on the two models for different heuristics using the balanced attitude.

and flattening time) and the predicate encapsulations are compared and evaluated. If an Automatic model got a time out during flattening or solving and has a predicate encapsulation that does not resemble the one of its corresponding Manual model, then the Automatic model is not considered during the comparison.

Note that the spacing, line-breaks, variable names, and possibly other syntax have been changed to improve the brevity and clarity of the Automatic models. Also note that the identification and encapsulation heuristic parameters were the same for all the tests.

6.4.1 Black Hole Patience Problem Comparison

In Table 6.3, we can see the runtimes, including flattening, of the models. The models not included in the table timed out during flattening on all instances. The flattening times were very similar across the instances for each model: only differing by a few tens of milliseconds. The arithmetic mean of the flattening time for the Bare model was 0.6 s, the mean for the Manual model was 0.9 s, the mean for the Argument Modesty model was 0.6 s, the mean for the Mix model was 0.8 s. As seen in the table, the models were solved similarly quickly: the Manual model is solved most quickly across all but the instances that all models were solved in under 40 ms; the Bare and Argument Modesty models are solved in strikingly similar times, hinting at their similarities; and the Mix model being solved more slowly than the Bare model (and the others) across all instances. Furthermore, the fifth column shows the runtimes of a model called Mix-Fix. This is the same model as the Mix model but where the unnecessary arguments are removed, as explained below. As can be seen, this fix improved the runtime; however, it is still far from being competitive with the Manual model. The arithmetic mean of the flattening time for the Mix-Fix model was 0.6 s.

In Figure 6.2, a snippet of the Manual model is shown for reference.

In Figure 6.3, a snippet of the Automatic model, using Argument Modesty, is shown. This model has very similar solve times to the Bare model (from Figure 2.10), and, as can be seen, the encapsulation is rather trivial. The constraint in the Bare model is x \[1\] == 1 and does not look like it contains any submodels worth presolving, because of its
<table>
<thead>
<tr>
<th>Instance</th>
<th>Model</th>
<th>Bare</th>
<th>Manual</th>
<th>Arg. Mod.</th>
<th>Mix</th>
<th>Mix-Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.dzn</td>
<td>10025</td>
<td>8660</td>
<td>10010</td>
<td>58966</td>
<td>58738</td>
<td></td>
</tr>
<tr>
<td>1.dzn</td>
<td>1258</td>
<td>1361</td>
<td>1264</td>
<td>4648</td>
<td>4450</td>
<td></td>
</tr>
<tr>
<td>2.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>3.dzn</td>
<td>946</td>
<td>1054</td>
<td>939</td>
<td>1412</td>
<td>1199</td>
<td></td>
</tr>
<tr>
<td>4.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>5.dzn</td>
<td>1956</td>
<td>1688</td>
<td>1954</td>
<td>5067</td>
<td>4813</td>
<td></td>
</tr>
<tr>
<td>6.dzn</td>
<td>637</td>
<td>892</td>
<td>638</td>
<td>815</td>
<td>649</td>
<td></td>
</tr>
<tr>
<td>7.dzn</td>
<td>1380</td>
<td>1364</td>
<td>1385</td>
<td>5092</td>
<td>4904</td>
<td></td>
</tr>
<tr>
<td>8.dzn</td>
<td>627</td>
<td>882</td>
<td>648</td>
<td>848</td>
<td>648</td>
<td></td>
</tr>
<tr>
<td>9.dzn</td>
<td>59488</td>
<td>33123</td>
<td>58739</td>
<td>107526</td>
<td>107439</td>
<td></td>
</tr>
<tr>
<td>10.dzn</td>
<td>653</td>
<td>884</td>
<td>644</td>
<td>966</td>
<td>777</td>
<td></td>
</tr>
<tr>
<td>11.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>12.dzn</td>
<td>4120</td>
<td>3091</td>
<td>4114</td>
<td>18875</td>
<td>18657</td>
<td></td>
</tr>
<tr>
<td>13.dzn</td>
<td>821</td>
<td>1039</td>
<td>812</td>
<td>1503</td>
<td>1302</td>
<td></td>
</tr>
<tr>
<td>14.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>15.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>16.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>17.dzn</td>
<td>648</td>
<td>883</td>
<td>638</td>
<td>848</td>
<td>658</td>
<td></td>
</tr>
<tr>
<td>18.dzn</td>
<td>25128</td>
<td>19340</td>
<td>25384</td>
<td>298791</td>
<td>298853</td>
<td></td>
</tr>
<tr>
<td>19.dzn</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>20.dzn</td>
<td>35289</td>
<td>29622</td>
<td>35643</td>
<td>342252</td>
<td>342490</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Solving runtime of five versions (Bare, Manual, Automatic using Argument Modesty, Automatic using Mix, Automatic using Mix with a manually performed fix) of the BHPP model. A breadth of $k = 100$ and the balanced attitude were used for the Automatic models. To denote a timeout during solving, we use ‘timeout’. The instances used here are exactly the instances listed at [https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models](https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models) under the black-hole/instances directory.

simplicity. This is unwanted behaviour left unfixed for time reasons; we do not see an easy fix for this, aside from always using the argument modesty heuristic in combination with some other heuristic.

In Figure 6.4, a snippet of the Automatic model, using Encapsulation Significance, is shown. As can be seen, this encapsulation bears some resemblance to the Manual model. Nevertheless, it did time out during flattening across all the instances. This timeout may be due to the bigger search space of using $x$ and $i$, instead of using the two variables $a$ and $b$ (respectively representing $x[i]$ and $x[i+1]$) as in the Manual model.

In Figure 6.5, a snippet of the Automatic model, now using Mix I, is shown. This also bears a resemblance to the Manual model; however, instead of having the subtraction inside the predicate, it is outside. Note that there are two unnecessary arguments in the predicate definition, $c$ and $d$: The encapsulated submodel comes from an unnecessary abstraction, the abstracted arguments $c$ and $d$ could be removed and replaced with the constants used in the predicate call without any problem, and the domain of the
% Consecutive cards match

constraint forall (i in 1..51) (adjacent(x[i], x[i+1]));

predicate adjacent(var 1..52: a, var 1..52: b) :: presolve(autotable) = ((a-b) in {13*i+1 | i in -4..3} union {13*i-1 | i in -3..4}) :: domain;

Figure 6.2: A snippet of the Manual model of the BHPP, showing the encapsulated submodel.

% Ace of Spades is first card
constraint firstcard(x[1]);

predicate firstcard(var 1..52: c) :: presolve(autotable) = c == 1;

Figure 6.3: A snippet of the Automatic model of the BHPP (using Argument Modesty), showing the encapsulated submodel.

argument ab could be shrunk by a large degree. That would most likely improve the solving speed tremendously, since these arguments also have unnecessarily big domains: int. Removing these arguments resulted, as can be seen in the table under Mix-Fix, in a slightly better model. However, the necessarily large domain of ab seems to be the main reason for this Automatic model to be so much worse than the Manual model. We do not see an easy fix for this problem; we discuss the problem in the future work section (Section 10.2). The problem of the large domain of ab most likely comes from the simplified implementation of the bounds function.

6.4.2 Block Party Metacube Problem Comparison

In Table 6.4, we see the runtimes, including flattening, of the models. The models not included in the table timed out under flattening on all instances, or, as in the case of the model produced with the Encapsulation Significance heuristic, is a faulty Automatic model that does not correspond to the Bare model. This bug has been neither fixed nor investigated because of time reasons. However, we do believe that it comes from the process of deriving types of the introduced parameters and argument variables (described in Section 5.5.1; discussed in Section 10.2). The flattening times were very similar across the instances for each model: only differing by a few tens of milliseconds. The flattening time for the Bare model was 0.2 s for all instances, the flattening time for the Manual model was also 0.2 s for all instances, and the arithmetic mean of the flattening times
% Consecutive cards match
constraint forall (i in 1..51) ( adjacent(x, i) );

predicate adjacent(array [1..52] of var 1..52: x, 1..51: i) :: presolve(autotable)
= ((x[i] - x[i + 1]) in {13*j+1 | j in -4..3} union {13*k-1 k in -3..4}) :: domain;

Figure 6.4: A snippet of the Automatic model of the BHPP (using Encapsulation Significance), showing the encapsulated submodel.

% Consecutive cards match
constraint forall (i in 1 .. 51) ( adjacent(x[i] - x[i+1], 13, 1) );

predicate adjacent(var int: ab, int: c, int: d) :: presolve(autotable)
= (ab in {c*j+d | j in -4..3} union {c*k-d | k in -3..4}) :: domain;

Figure 6.5: A snippet of the Automatic model of the BHPP (using Mix I), showing the encapsulated submodel.

for the Automatic model, using Argument Modesty, was 5.6 s, where the vast majority of the flattening time consisted of presolving the annotated predicate. As seen in the table, the Manual model is solved the quickest across all instances, the Bare model comes second but is solved much slower than the Manual model across all instances, and the Automatic model is solved the slowest across all instances but with very similar times to the Bare model. These are not ideal results, and we have no real clue as to why these are the results. This has not been investigated because of time reasons.

We will not compare and show the contents of the generated MiniZinc models for time reasons.

7 Discussion

Many of the questions in the problem statement (Section 1.1) are seen to be answered rather well in this thesis (Section 7.1). We discuss the drawbacks, limitations, and other aspects of MZL (Section 7.2), the theory and implementation (Section 7.3), and the evaluation (Section 7.4).
Table 6.4: Solving runtime of three versions (Bare, Manual, and Automatic using Search Space Modesty) of the BPMP model. A breadth of $k = 100$ and the balanced attitude were used for the Automatic model. The instances used here are exactly the instances listed at https://github.com/Dekker1/MiniZinc-Auto-Tabling-Models under the block-party/instances directory.

<table>
<thead>
<tr>
<th>Instance \ Model</th>
<th>Bare</th>
<th>Manual</th>
<th>S. Spa. Mod.</th>
</tr>
</thead>
<tbody>
<tr>
<td>412.dzn</td>
<td>15439</td>
<td>3420</td>
<td>17989</td>
</tr>
<tr>
<td>1176.dzn</td>
<td>13193</td>
<td>2853</td>
<td>15224</td>
</tr>
<tr>
<td>1266.dzn</td>
<td>11972</td>
<td>2522</td>
<td>13666</td>
</tr>
<tr>
<td>2254.dzn</td>
<td>8066</td>
<td>1698</td>
<td>9272</td>
</tr>
<tr>
<td>3237.dzn</td>
<td>14505</td>
<td>3125</td>
<td>16765</td>
</tr>
<tr>
<td>3345.dzn</td>
<td>7292</td>
<td>1612</td>
<td>8462</td>
</tr>
<tr>
<td>4629.dzn</td>
<td>15923</td>
<td>3476</td>
<td>18303</td>
</tr>
<tr>
<td>7059.dzn</td>
<td>6218</td>
<td>1271</td>
<td>7152</td>
</tr>
<tr>
<td>7324.dzn</td>
<td>13013</td>
<td>2761</td>
<td>15058</td>
</tr>
<tr>
<td>7326.dzn</td>
<td>8971</td>
<td>1894</td>
<td>10331</td>
</tr>
<tr>
<td>7444.dzn</td>
<td>9167</td>
<td>2057</td>
<td>10637</td>
</tr>
<tr>
<td>7661.dzn</td>
<td>7392</td>
<td>1671</td>
<td>8673</td>
</tr>
<tr>
<td>7750.dzn</td>
<td>21181</td>
<td>4758</td>
<td>24096</td>
</tr>
<tr>
<td>8848.dzn</td>
<td>20046</td>
<td>4519</td>
<td>23161</td>
</tr>
</tbody>
</table>

7.1 Discussion of the Research Questions

We will revisit the questions posted in the introduction (Section 1.1) and see how well this thesis answered these.

**What are the submodels of a model, and how do we find them?** In Section 4.1 we present a method for identifying all or a subset (with the most promising ones) of the submodels in a model. This method is then realised as described in Section 5.5.1.

Because of time reasons, no evaluation of the actual submodel identification was done in this thesis; this, however, could be a very interesting thing to look at. Although it is believed that the theory presented in this thesis can identify all the submodels of a MiniZinc model (or, at least all submodels that MZL allows for), this is, again because of time reasons, not tested or proved throughout this thesis.

**What makes a submodel profitably presolvable, and are there some general rules that can be systematised?** Using the findings of Dekker et al. and guesswork, we have presented six base heuristics and one compound heuristic. These seven heuristics have been shown to produce models similar to the manually encapsulated ones.

However, more testing and evaluation is needed to fine-tune a compound heuristic that finds profitably presolvable submodels more often than the already presented heuristics.
What are the most profitably presolvable submodels of a model? Using the heuristics we can evaluate all the submodels identified and in that way find the best one.

One problem with this approach is that, although it is believed to be a rare case, it is possible that an $\infty$-breadth SIG has to be created in order to truly identify a best submodel, according to the heuristics. This comes from the fact that the breadth can have an impact on the Encapsulation Significance heuristic.

It is also hard to predict whether or not a submodel, when presolved, will improve nearly all or a majority of the interesting instances of a MiniZinc model.

How can we refactor a MiniZinc model to encapsulate a given submodel? This process is rather straightforward and we have shown methods for automatically rewriting a model to encapsulate a given submodel. Note, however, that generalisations are not a part of this solution, for time reasons. It is unclear how useful the addition of generalisations will be, but we believe that it is insignificant and only useful in a few cases.

Perhaps it would have been better to base the refactoring part of the encapsulation process on prior concepts and techniques in the field of code refactoring [13].

7.2 Discussion of our MiniZinc Logic

MZL is lacking in many aspects, even for the purposes of this thesis, since it is not a complete representation of the MiniZinc language. For example, parameter variable and decision variable items, in MiniZinc, may be assigned some expression, and this expression may contain submodels, but these expressions are not represented as terms or formulas in MZL, but by some external rule of the logic’s interpretations.

The type system in this thesis is not extensively defined for time reasons. But since a thorough type system is not necessary for the purposes of this thesis, this was overlooked.

7.3 Discussion of the Theory and Implementation

One problem with heuristics is that it is impossible to derive exactly how everything in a model is intended to be used.

One drawback with the theory and implementation is that we cannot go into the predicate and function definitions.

The current implementation contains many hacks and ugly tricks to solve some of the problems, for which there most likely are much more elegant, efficient, and perhaps even effective solutions. For example, in the case of equality between two MiniZinc expressions, a recursive formula is used that only detects some commutativity in the expressions. Furthermore, MiniZinc contains no information about what functions and predicates are commutative or associative; it is thus extremely hard to know if two expressions, using functions and predicates, are equivalent in some cases.

The tool has some limitations and known bugs. For example, it cannot handle the shadowing of names.
Another weird quirk is that the lower bound is not always less than the upper bound in scores. This partly comes from the flipping of some of the scores, which simply subtracts the score from the maximal number in the normalised range, but also from the fact that the calculation of finding upper and lower bounds are oversimplified. Furthermore, this problem also comes from the fact that the operations for scores are implemented incorrectly. This was realised too late and not fixed due to time reasons.

7.4 Discussion of the Evaluation

Since the evaluation of the implementation is scarce, it makes it hard to draw any conclusions at this point. This of course limits the impact that this work can have at this point. The scarce evaluation is due to time constraints.

8 Related Work

While there is no work really similar to this thesis, there are some that share enough similarities that they are worth mentioning here. Note that the related work of Dekker et al. [7, 8] is explained in Section 2.5 and is the main inspiration for this thesis.

There are prior tools that analyse MiniZinc models in order to suggest improvements or ways of solving a model.

One example is the Globalizer [14], which can suggest improvements to MiniZinc models. It analyses subsets of constraints in a MiniZinc model in order to see if they can be replaced with a more general constraint (in this case, global constraints; hence the name Globalizer) that has a better inference algorithm. This is done by splitting a model into submodels and then instantiating each of the resulting submodels into a set of submodel instances. From such a set, candidate constraints are generated by sampling the solution space of both the set and the candidate constraints. The Globalizer makes suggestions to the modeller with an associated confidence, which comes from the similarity of the search space of a given set of submodel instances and the search space of the suggested candidate constraint for this submodel, and has been shown to aid both novice and experienced modellers.

Another example is the sunny-cp2 portfolio solver [15], which extracts dynamic features [16] from a MiniZinc model and instance data in order to determine which solver of which technology seems best suited for the model and instance. The selection of solver and technology is done via learning algorithms based on those features. Note that sunny-cp2 does not use any extracted information to improve the model.

9 Conclusion

We have presented techniques and theory for identifying submodels from a MiniZinc model, for evaluating a given submodel for the expected profitability of dynamically presolving that submodel, and for automatically rewriting a MiniZinc model so that a given submodel gets encapsulated within a predicate definition and annotated for
presolving. We have implemented and explained the design of a tool that works directly on MiniZinc models and applies these techniques and concepts. In order to explain and present these concepts, an incomplete, albeit useful, formalisation of the MiniZinc language has been presented.

We have shown that these techniques can produce models with encapsulations similar to manual encapsulations that are profitably presolvable, performed by expert modellers, and that the encapsulations produced can reduce the total runtime when compared to the bare models without the encapsulation and presolving.

It is hard to draw conclusions yet since the evaluation has not been thorough enough: Not enough combinations of heuristics and other settings have been tested and evaluated. Even though the evaluation is incomplete, and even though impressive results have been sparse, we believe that the infrastructure and concepts are useful and hold great potential: Given the right combinations of heuristics and settings, profitably presolvable encapsulations can be automatically found. We believe that the combination of heuristics is crucial for finding profitably presolvable submodels.

10 Future Work

Some ideas regarding MZL are presented in Section 10.1. Except for thoroughly testing the tool and fixing all its remaining bugs, there are some other interesting ideas worth pursuing regarding the theory and implementation. These ideas are presented here in Section 10.2.

10.1 Future Work on our MiniZinc Logic

MZL could be extended in such a way that it could capture all Boolean expressions within a MiniZinc model, as not every part of MiniZinc code gets turned into formulas or terms in MZL, and some are unreachable in a SIG (predicate and function definitions, and the objective term).

MZL may be interesting for other purposes than those in this thesis. For these other purposes, the type system most likely needs to be revisited and defined more thoroughly.

10.2 Future Work on the Theory and Implementation

One task is to evaluate more, and try to find and fine-tune better combinations of the heuristics, and perhaps come up with brand-new ones. This will make the potential of the concepts and theory in this thesis clearer and reveal how much potential there actually is. A good start could be to test the tool across more models, on more combinations of settings (attitude and breadth), and with more MiniZinc backends (i.e., solvers that can read MiniZinc models). Constructing a lot more combinations of heuristics and evaluate each of them to find the generally good and the generally bad heuristics could also be a good start.
Another task to consider is to make it possible to encapsulate submodels within the objective function as well. Although we believe this to be rarely useful, the objective function may include formulas, which of course can be encapsulated.

Methods for optimising the vertices of a SIG are an interesting topic and could improve the accuracy of finding profitably presolvable submodels to a large degree. This is interesting since much of the automatically produced encapsulations have unnecessary arguments or have unnecessarily large domains for their arguments. Shrinking these domains and removing these arguments will most likely improve the automatically produced models tremendously. Two approaches could be taken here: either first improving the submodels of all the vertices of the SIG and then finding a best one, or first finding a best one and then improving that one.

One weakness with both the theory and the implementation is that a conjunction of model-level constraints cannot be encapsulated within a predicate. Making this possible seems like a rather trivial task compared to some of the other ideas of future work presented here.

Another idea is to make it possible to annotate the model for influencing and passing information to the encapsulation tool. Perhaps this could include information about whether a defined function is associative or commutative (or both), or whether one parameter will most likely be less than another.

In the implementation of the submodel identification process, the types of introduced parameters (when creating subformulas) and introduced argument variables (when creating abstractions) need to be derived. This process is very crudely implemented and needs to be thoroughly tested and greatly improved upon. We believe that bug encountered (in Section 6.4.2) when running the block party metacube model comes from this process.

Also, in the implementation, the bounds function is very simplified, giving many of the arguments of the encapsulated submodels unnecessarily large domains. This can tremendously slow down the presolving process, as seen in the evaluation section (Section 6).

Finally, extending the theory and implementation to fully support generalisations is another idea that is worth pursuing. This idea is of rather low priority, however, since we believe that the addition of generalisations is insignificant to the submodel identification process, as mentioned.

A Black Hole MiniZinc Model Licence

The licence of the black hole patience problem MiniZinc model used throughout this thesis is shown below:

The model of the problem is taken from “Search in the Patience Game ‘Black Hole’”, by Ian P. Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M. Smith, and S. Armagan Tarim. It only implements the basic model. The instances are generated by the black hole
patience model in the Gecode distribution.

This model uses the following global constraints - inverse - table

Main authors: Mikael Zayenz Lagerkvist (lagerkvist@gecode.org)

Copyright: Mikael Zayenz Lagerkvist, 2009

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B Installation Instructions

The implementation, with installation instructions, can be found on GitHub (https://github.com/aekh/mzn-encaps). The tag ‘thesis’ refers to the version used for this thesis.

C A Theory of Constraint Satisfaction Families

A theory was created in order to extend and generalise the CSP and the COP to reflect the power of constraint modelling in a mathematical sense. It is a formalisation of families of CSP and COP instances. This theory was originally created for the purpose of submodel (and subproblem) identification; although it was abandoned in favour of the theory presented in Section 4, it is still interesting theoretically, and includes useful concepts and ideas.

The concepts and definitions regarding constraint satisfaction families are shown and explained in Appendix C.1 We define and explain the link graph in Appendix C.2, which captures the structural properties of constraint satisfaction families. For the purpose of subproblem identification, this theory is incomplete and has some weaknesses; we discuss these and present ideas for future work, in Appendix C.3.
C.1 Introduction and Definitions

Many of the concepts presented in this section will be reminiscent of some of the background concepts presented in Section 2.

In constraint modelling—and in particular, MiniZinc—models can be expressed with respect to one or more parameters. As a result, MiniZinc models do not represent a CSP instance, but rather a family of CSP instances: only when given instance data, a CSP instance is obtained.

The theory presented is abstract and independent of MiniZinc and any other constraint-based modelling language.

Consider, again, the Sudoku problem presented in Section 2.1. The Sudoku problem will be examined in the context of constraint problems and constraint-based modelling in order to fully convey the underlying theory of this section. We begin by examining the instance data of Sudoku. The clues of the Sudoku denote the instance data, and some sets of clues form unsatisfiable Sudoku puzzles. All the instances of a combinatorial problem can be represented by a set of objects, where each object represents a unique instance of the problem. In Sudoku, each set of possible combination of clues, even unsatisfiable ones and the Sudoku puzzle with no clues, can be represented by such an object. We call these objects instance objects and the set of all instance objects of a given combinatorial problem is called the instance universe of that problem. Formally, they are defined as follows:

Definition C.1 (Instance Data). Given a combinatorial problem, its instance universe is a countable (possibly infinite) non-empty set \( \Delta \) of instance objects, where each instance object \( \delta \in \Delta \) is an abstract object that represents a unique instance of the problem. If \( \Delta \) has exactly one instance object, then \( \Delta \) is called the trivial instance universe. Every model of a combinatorial problem has the same instance universe as the combinatorial problem itself. □

If a combinatorial problem has the trivial instance universe, then it is considered equivalent to a CSP instance. For example, consider the Sudoku problem with the clues of Figure 2.1 as a problem (i.e., the clues are a part of the problem, not the instance data): it has only one instance because no parameters can be changed. Hence that problem has a trivial instance universe and can, in fact, be represented directly by a CSP instance. In a model representation of a combinatorial problem, the instance data of the problem are described by a set of parameters; where each combination of parameter values represents a unique instance object and all these combinations together form the instance universe.

A combinatorial problem can be represented by a family of CSP instances, which is a function that maps each instance object to a unique CSP instance. Each such family is called a constraint satisfaction family (CSF). For example, the Sudoku problem can be represented by a CSF that maps each instance object (which represents how the board looks like) to the actual Sudoku puzzle with that board, along with the necessary decision variables, variable domain mapping, and constraints (i.e., the CSP instance of that board). Note that a combinatorial problem may have multiple equivalent CSFs. If two CSFs, for every instance object, map to equivalent CSP instances, then we say that
the two CSFs are *equivalent*. As we unpack this concept of CSFs, we will see how closely it relates to models of combinatorial problems.

In order to get some easily identifiable structural properties of CSFs, a more concrete description than just an abstract mapping is necessary. We divide a CSF into three parts: a *decision variable family* (DVF), which maps instance objects to a set of decision variables; a *variable domain family*, which maps instance objects to a variable domain mapping of the decision variables of that instance; and, lastly, a *constraint family* (CF), which maps instance data to a set of constraints on the decision variables of that instance.

Before CSFs can be defined in detail, we need to define formally the set of all constraints of a set of decision variables:

**Definition C.2.** Let $\mathbb{DEC}$ denote the chosen universe of decision variables. Now, the set of all possible constraints on $\mathbb{DEC}$ is defined as

$$\mathbb{CON} = \left\{ (S, R) \mid R \subseteq \mathbb{Z}^k, \ S \subset \mathbb{DEC}, \ \text{and} \ |S| = k \right\}.$$  

Now, CSFs can be formally defined (recall that $\mathcal{P}$ is used to denote the powerset of a set):

**Definition C.3 (CSF).** A *constraint satisfaction family* (CSF) is a family of CSP instances encoded as a 4-tuple $C = \langle \Delta, V, D, C \rangle$, where $\Delta$ is an instance universe and:

- $V$ is a *decision variable family* (DVF) and is a function $V : \Delta \to \mathcal{P}(\mathbb{DEC})$ that maps every instance object $\delta \in \Delta$ to a finite set of decision variables $V(\delta) \subseteq \mathbb{DEC}$.

- $D$ is a *variable domain family* and is a function $D : \Delta \to (\mathbb{DEC} \to \mathcal{P}(\mathbb{Z}))$ that maps every instance object $\delta \in \Delta$ to a variable domain mapping $D_\delta$. Furthermore, for every instance object $\delta \in \Delta$, the map $D_\delta$ associates a (finite) variable domain to each decision variable in the family $V(\delta)$ of decision variables (i.e., $D_\delta(v) \subseteq \mathbb{Z}$).

- $C$ is a *constraint family* (CF) and is a function $C : \Delta \to \mathcal{P}(\mathbb{CON})$ that maps every instance object $\delta \in \Delta$ to a finite set of constraints $C(\delta) \subseteq \mathbb{CON}$. For every instance object $\delta \in \Delta$, each constraint $C \in C(\delta)$ constrains some subset $\text{scope}(C) \subseteq V(\delta)$ of decision variables.

- The CSF $C$ is a map $C : \Delta \to \text{CSP}$ from instance data to CSP instances. For every instance object $\delta \in \Delta$, we denote $C(\delta) = \langle V(\delta), D(\delta), C(\delta) \rangle$ to be the instance of $C$ under $\delta$. Lastly, given two instance objects $\delta_1, \delta_2 \in \Delta$, if $C(\delta_1) = C(\delta_2)$, then we consider $\delta_1 = \delta_2$ under $C$.  

In a similar fashion as for CSP instances, we have solutions to CSFs, called solution families:

**Definition C.4 (Solution Family).** A *solution family* to a CSF $C = \langle \Delta, V, D, C \rangle$ is a family of functions $S : \Delta \to (\mathbb{DEC} \to \mathbb{Z})$ that maps every instance object $\delta \in \Delta$ to a solution $S(\delta) : V(\delta) \to D_\delta[V(\delta)]$ of the CSP instance $C(\delta)$.
Now, we will see how the Sudoku problem can be encoded as a CSF. Note that this is one of many possible encodings. Let \( \langle \Delta, V, D, C \rangle \) be a CSF representing the Sudoku problem explained in Section 2.1. The instance universe \( \Delta \) contains elements that each is associated with a unique set of clues, together forming all possible sets of clues. Let \( \delta \in \Delta \) be an arbitrary instance object. The DVF \( V \) maps every instance object to a set of 81 decision variables, each associated with a cell of its own, and is defined as
\[
V(\delta) = \{v_{11}, v_{12}, \ldots, v_{19}, v_{21}, v_{22}, \ldots, v_{29}, \ldots, v_{91}, \ldots, v_{99}\}.
\]
The variable domain family \( D \) maps to a domain function that maps every decision variable \( v \in V(\delta) \) is defined as
\[
D_\delta(v) = \begin{cases} 
\{k\} & \text{if, in } \delta, \text{ the cell of } v \text{ has a clue of value } k, \\
\{1,2,\ldots,9\} & \text{otherwise.}
\end{cases}
\]
The CF captures the three rules and can be split into three (in this case, but not necessarily, disjoint) sets for every instance object as
\[
C(\delta) = \{C_{r_1}, C_{r_2}, \ldots, C_{r_9}\} \cup \{C_{c_1}, C_{c_2}, \ldots, C_{c_9}\} \cup \{C_{11}, C_{12}, C_{13}, C_{21}, \ldots, C_{33}\}.
\]
The first, second, and third set represent the row constraints, the column constraints, and the region constraints, respectively. In total there are 27 constraints in this formulation, for every instance object. Let \( \text{perm}(X) \) denote the set of all permutations (in tuple form) of a given set \( X \). Now, each of the constraints \( C_{r_1}, C_{r_2}, C_{ij} \in C(\delta) \) can be defined as
\[
C_{r_i} = \langle \{v_{11}, v_{12}, \ldots, v_{19}\}, \text{perm}\{\{1,2,\ldots,9\}\} \rangle, \\
C_{c_i} = \langle \{v_{1i}, v_{2i}, \ldots, v_{9i}\}, \text{perm}\{\{1,2,\ldots,9\}\} \rangle, \text{ and} \\
C_{ij} = \langle \{v_{xy} \mid x \in \{3i, 3i - 1, 3i - 2\} \text{ and } y \in \{3j, 3j - 1, 3j - 2\}\}, \text{perm}\{\{1,2,\ldots,9\}\} \rangle.
\]
Note that the CSF will map to the same row, column, and region constraints for every instance object. As a result, this splitting of the CF is consistent across all instances. Often it is the case, however, that the set of constraints differs between instances.

Since the above way of reasoning on CFs, DVFs, and their parts is tedious, we introduce some concepts below to allow a better and clearer reasoning. Subimage functions are a way to reason about mappings to sets:

**Definition C.5 (Subimage Functions).** Let \( f, g : X \to \mathcal{P}(Y) \) be two functions. The function \( f \) is a subimage function of \( g \) (denoted \( f \subseteq g \)) if for every \( x \in X \) the set \( f(x) \) is included in \( g(x) \), that is, \( f(x) \subseteq g(x) \). Conversely, \( g \) is a superimage function of \( f \) in that case. We denote the set of all subimage functions of \( f \) as \( f^\subseteq = \{g \mid g(x) \subseteq f(x), \text{ for every } x \in X\} \).

For a function \( f : X \to \mathcal{P}(Y) \), there exist \( 2^{\sum_{x \in X}|f(x)|} \) different subimage functions of \( f \), including \( f \) itself and the function \( f_0 : X \to \{\emptyset\} \).

Given two sets \( X \) and \( Y \), we get the partial ordering, and complete lattice, \( (X \to \mathcal{P}(Y), \subseteq) \) of subimage functions.
Note that, given a CSF \( (\Delta, V, D, C) \), every subimage function of \( V \) is also a DVF and every subimage function of \( C \) is also a CF. However, most of the time the terms \( \text{sub-DVF} \) and \( \text{sub-CF} \) will be used in these situations for clarity. The empty sub-DVF and the empty sub-CF, which both map to the empty set for every instance object, will be ignored and left unused throughout this appendix because they are deemed uninteresting.

Image-union functions are a way to combine, into a new function, multiple functions that maps to sets:

**Definition C.6 (Image-union Functions).** Let \( f : X \rightarrow \mathcal{P}(Y) \) and \( g : X \rightarrow \mathcal{P}(Y) \) be two functions. The *image-union function* of \( f \) and \( g \) is the function \( f \sqcup g : X \rightarrow \mathcal{P}(Y) \) that maps every element \( x \in X \) to the union \( (f \sqcup g)(x) = f(x) \cup g(x) \).

Let \( F \subseteq X \rightarrow \mathcal{P}(Y) \) be a set of functions. We define the image-union function of \( F \) as follows:

\[
\forall x \in X : \ y \in \bigsqcup F(x) \iff \exists f \in F, \ y \in f(x) .
\]

As a result, the image-union function of the empty set is the empty subimage function, which is also the identity element of \( \sqcup \) over \( X \rightarrow \mathcal{P}(Y) \). The operation \( \sqcup \) is associative, commutative, and is closed on \( X \rightarrow \mathcal{P}(Y) \). Hence, \( \sqcup \) forms a commutative monoid. In the complete lattice \( (X \rightarrow \mathcal{P}(Y), \sqsubseteq) \) of subimage functions, for each subset \( S \subseteq X \rightarrow \mathcal{P}(Y) \) of functions, the supremum of \( S \) is the image-union function \( \bigsqcup S \).

Image-union functions allow reasoning on parts of a CF or DVF. For example, the CF of the presented Sudoku CSF can be split in the following way:

\[
C = C_r \sqcup C_c \sqcup C_R
\]

where \( C_r \) represents the set of row constraints, the sub-CF \( C_c \) represents the set of column constraints, and \( C_R \) represents the set of region constraints. That is:

\[
\begin{align*}
C_r(\delta) &= \{C_{r1}, C_{r2}, \ldots, C_{r9}\} \\
C_c(\delta) &= \{C_{c1}, C_{c2}, \ldots, C_{c9}\} \\
C_R(\delta) &= \{C_{11}, C_{12}, C_{13}, C_{21}, \ldots, C_{33}\}
\end{align*}
\]

Notice that this splitting of the CF is completely arbitrary and can be done in many other ways. The only criterion is that the image-union function of all the sub-CFs is equal to the CF of the CSF.

In a similar manner as scopes are defined for constraints in the CSP, scope families are defined for CSFs. Instead of just a set of decision variables (which does not really exist in CSFs), a sub-DVF is used to capture the union of every scope of every constraint that the sub-CF produces given the corresponding instance object.

Before defining scope families formally, we will see how these relates to the Sudoku CSF that has been described in this section. For example, the scope family for \( C_r \) is \( \delta \mapsto V(\delta) \), since all the decision variables are used for every instance object. This is the scope family for the other two sub-CFs as well, for the same reason.

We can further split each of these three sub-CFs into nine sub-CFs of its own (i.e., 27 sub-CFs in total, one for each constraint). In that case, the scope families look different.
We would have a CF for the first row only, call it $C_{r_1}$. The scope family for $C_{r_1}$ is the sub-DVF $\delta \mapsto \{v_{11}, v_{12}, \ldots, v_{19}\}$. In this case, we always only constrain the first row of the grid. Again, the scope family would look more complicated if the DVF did not create the same set of decision variables for every instance object. Formally, scope families are defined as follows:

**Definition C.7** (Scope Family). Let $\langle \Delta, V, D, C \rangle$ be a CSF and $C' \subseteq C$ be a sub-CF. The constraint scope family (or simply scope family) of the sub-CF $C'$ is the function $\text{csf}_{C'} : \Delta \rightarrow \mathcal{P}(\text{DEC})$ that maps every instance object $\delta \in \Delta$ to the union of the constraint scopes $\text{csf}_{C'}(\delta) = \bigcup \text{scope}(C(\delta))$. It is a subimage function of $V$. □

Here it is interesting to see how this is extended to the COP. The only difference is that we now have an objective function for every instance. We only get one extra part, namely an objective function family.

**Definition C.8** (COF). A constrained optimisation family (COF) is a family of COP instances encoded as a 5-tuple $\langle \Delta, V, D, C, F \rangle$, where $\langle \Delta, V, D, C \rangle$ is a CSF, and $F$ is an objective function family. An objective function family is a map from instance data to objective functions where $F(\delta)$ is defined on a subset of $V(\delta)$, for every instance $\delta \in \Delta$. If $F$ maps to a constant function for some instance object, then that problem instance is a CSP instance, and if $F$ always maps to a constant function, then $C$ is a CSF also. □

As a COP instance has an objective scope, a COF similarly has an objective scope family, defined next:

**Definition C.9** (Objective Scope Family). Let $\langle \Delta, V, D, C, F \rangle$ be a COF. The objective scope family of the objective function family $F$ is the subimage function $\text{csf}_F : \Delta \rightarrow \mathcal{P}(\text{DEC})$ that maps every instance object $\delta \in \Delta$ to the scope $\text{csf}_F(\delta) = \text{scope}[F(\delta)]$ of the objective function. □

Note that an objective scope family is a sub-DVF.

### C.2 Constraint Graph-like Structures for Constraint Satisfaction Families

Each CSP instance can be represented and visualised as a graph, called the constraint graph of the CSP instance. In the case of CSFs, we have, as a result, one constraint graph for each instance object of CSF.

A CSF cannot be represented as a constraint graph in the same fashion as CSP instances, since the constraints and decision variables depend on the instance object.

A link graph contains a lot of structural information about a CSF. Before describing link graphs, links needs to be understood.

For every two sub-CFs we have something called the link between them. Take two sub-CFs of the Sudoku CSF, say the first row constraint $C_{r_1}$ and the first column constraint $C_{c_1}$. These two sub-CFs intersect on decision variable $v_{11}$ for every instance object. Hence, we get the link $\delta \mapsto \{v_{11}\}$. A link is essentially a function that maps
every instance object to the set of decision variables that the resulting constraints, from each sub-CF, overlap on in that instance. As a result, the link between two sub-CFs is a sub-DVF.

A link is a way to see where two constraints intersect across instance data. The main idea here is to use a structure similar to the dual graph of a constraint graph in combination with links.

Before we can formally define links, we need another tool to reason on functions: image-intersection functions.

**Definition C.10 (Image-intersection Functions).** Let $f : X \to \mathcal{P}(Y)$ and $g : X \to \mathcal{P}(Y)$ be two functions. The image-intersection function of $f$ and $g$ is the function $f \cap g : X \to \mathcal{P}(Y)$ that maps every element $x \in X$ to the intersection $(f \cap g)(x) = f(x) \cap g(x)$.

Let $F \subseteq X \to \mathcal{P}(Y)$ be a set of functions. We define the image-intersection function of $F$ as follows:

$$\forall x \in X : y \in \bigcap F(x) \iff \forall f \in F, y \in f(x).$$

As a result, the image-intersection function of the empty set is the function that maps every element to $Y$, which is also the identity element of $\cap$ over $X \to \mathcal{P}(Y)$. The operation $\cap$ is commutative and is closed on $X \to \mathcal{P}(Y)$. In the complete lattice $(X \to \mathcal{P}(Y), \subseteq)$ of subimage functions, for each nonempty subset $S \subseteq X \to \mathcal{P}(Y)$ of functions, the infimum of $S$ is the image-intersection function $\bigcap S$.

Now the formal definition of links can be presented:

**Definition C.11 (Link).** Let $\langle \Delta, V, D, C \rangle$ be a CSF and $C_1, C_2 \subseteq C$ be two sub-CFs. The link between $C_1$ and $C_2$ is the image-intersection function $\text{link}_{C_1}^{C_2} = (\text{csf}_{C_1} \cap \text{csf}_{C_2}) : \Delta \to \mathcal{P}(\text{DEC})$ that maps every instance object $\delta \in \Delta$ to the intersection of the unions of their constraint scopes: $\text{link}_{C_1}^{C_2}(\delta) = (\text{csf}_{C_1} \cap \text{csf}_{C_2})(\delta) = \bigcup \text{scope}[C_1(\delta)] \cap \bigcup \text{scope}[C_2(\delta)]$.

If the union of the image $\bigcup \text{link}_{C_1}^{C_2}[\Delta]$ is the empty set, then $C_1$ and $C_2$ are said to be unlinked, else they are said to be linked.

Some sub-CFs may overlap in only some of the problem instances while others overlap in every problem instance.

In order to create a link graph of a CSF, we need to decide what sub-CFs to use and connect with links. We call these parts CF components.

Before we can formally define CF components, we need yet another tool to reason on functions, namely image covers, defined next:

**Definition C.12 (Image cover).** Let $f : X \to \mathcal{P}(Y)$ be a function and let $G \subseteq f \subseteq f$ be a set of subimage functions of $f$ (i.e., $g \subseteq f$, for every $g \in G$). The set $G$ is an image cover of $f$ if the image-union function of $G$ is $f$, that is, $\bigcup G = f$.

Every set of functions from $X$ to $\mathcal{P}(Y)$ is an image cover of its image-union function.

Using image covers, we can now formally define the concept of CF components and CF componentisation:
Definition C.13 (CF Components). Let \( C \) be a CF and \( C \subseteq V^C \) be an image cover of \( C \). Whenever we limit the construction of subimage functions of \( C \) to be solely functions chosen from the set \( \{ \bigsqcup x \mid x \in P(C) \} \) of image-union functions of the subsets of \( C \), then we call \( C \) a CF componentisation of \( C \) and its elements CF components. □

In Figure C.1, we see how a link graph can look like. Each vertex of a link graph represents some CF component and each edge represents the link between the two CF components its incident vertices represent. Formally, link graphs are defined as follows:

Definition C.14 (Link Graph). Let \( \mathcal{E} = \langle \Delta, V, D, C \rangle \) be a CSF and \( C \) be a CF componentisation of \( C \). A link graph of \( C \) is a tuple \( \langle G, \kappa, \lambda \rangle \), where \( G = (V,E) \) is an undirected graph, \( \kappa : V \to C \) is a bijective function that maps each vertex \( v \in V \) to a unique sub-CF \( \kappa(v) \in C \), and \( \lambda : E \to V^C \) is a function that maps each edge \( e = (u,v) \in E \) to the link \( \lambda(e) = \text{link}_{\kappa(v)}^{\kappa(u)} \) of the sub-CFs corresponding to its incident vertices, such that there exists an edge \( e = (u,v) \) if and only if \( \kappa(u) \) and \( \kappa(v) \) are linked, that is, \( \bigcup \text{link}_{\kappa(v)}^{\kappa(u)}[\Delta] \neq \emptyset \). □

Theorem 1. For every CSF and for each of its CF componentisations, there exists exactly one corresponding link graph up to isomorphism.

Proof. Let \( \mathcal{E} = \langle \Delta, V, D, C \rangle \) be a CSF, the set \( C \) be a CF componentisation of \( C \), and \( \langle G_1, \kappa_1, \lambda_1 \rangle \) be a link graph of \( \mathcal{E} \) with \( C \), where \( G_1 = (V_1, E_1) \) is a graph. Let \( \langle G_2, \kappa_2, \lambda_2 \rangle \) be another link graph of \( \mathcal{E} \) with \( C \), where \( G_2 = (V_2, E_2) \) is a graph.

First, by definition, \( \kappa_1 \) and \( \kappa_2 \) are both bijections from \( V_1 \) onto \( C \) and \( V_2 \) onto \( C \), respectively; thus, the equality \( |V_1| = |V_2| \) holds. Hence, we have a bijection \( f : V_1 \to V_2 \) such that, for every \( v \in V_1 \), the equality \( \kappa_1(v) = \kappa_2(f(v)) \) holds. Furthermore, for every \( (u,v) \in E_1 \), we have that \( \kappa_1(u) \) and \( \kappa_1(v) \) are linked, by definition. Hence, \( \kappa_2(f(u)) \) and \( \kappa_2(f(v)) \) are linked; thus, by definition, the membership \( (f(u), f(v)) \in E_2 \) holds. As a result, \( G_1 \) and \( G_2 \) are isomorphic, witnessed by the bijection \( f \). □
C.3 Discussion and Future Work

One big problem with this theory is that it cannot encapsulate parts of a list comprehension, which is a very limiting problem. This comes from the way that subproblems are defined in this theory. This problem could potentially be fixed if some way of connecting this theory to MZL was found. One idea here is that every encapsulation of a submodel in MZL rewrites its link graph, since it will map to different FlatZinc models. Another idea is that each vertex in the link graph is, by itself, a link graph representing the MiniZinc expressions inside.

One useful thing in this theory is that the link graph could be used to define the treewidth of a constraint model, perhaps as a function of its parameters or with lower and upper bounds. If a connection between MZL and this theory is established, then this treewidth calculation could be used as an additional heuristic for evaluating submodels to encapsulate.

References


