



MONASH University

Making the Most of Structure in Constraint Models

by

Kevin Leo

BSc Computer Science (Hons)

A thesis submitted for the degree of Doctor of Philosophy at
Monash University in 2017
Caulfield School of Information Technology

Supervisors:

Dr. Guido Tack

Prof. Maria Garcia de la Banda

© Kevin Leo 2017

Contents

List of Tables	vii
List of Figures	viii
Abstract	x
Acknowledgments	xii
1 Introduction	1
1.1 The Modelling, Compilation, and Solving Process	3
1.2 The Importance of Good Models	7
1.3 Contributions	12
1.4 Structure of the Thesis	13
2 Background	14
2.1 Introduction	14
2.2 Combinatorial Problems	14
2.3 Solving Combinatorial Problems	15
2.3.1 Linear Programming	15
2.3.2 Mixed-Integer Programming	16
2.3.3 Boolean Satisfiability	20
2.3.4 Constraint Programming	22
2.4 High-Level Modelling	26
2.4.1 MiniZinc by Example	27
2.4.2 Grammar of a Simplified MiniZinc	28
2.5 Compilation	30
2.5.1 Compiling MiniZinc Instances to Programs: an Overview	30
2.5.2 Compiling MiniZinc Instances to Programs: a Procedure	33
2.5.3 Further Compilation Techniques	39
2.6 Model Structure	40
2.6.1 Losing Structure	41
2.6.2 Recovering Structure	42
2.7 Using Structure	42
2.7.1 Program Level	43
2.7.2 Instance Level	44

2.7.3	Model Level	46
2.8	Conclusion	47
3	A Framework for Model Structure	48
3.1	Introduction	48
3.2	The New Framework	49
3.3	Structure Sharing	50
3.3.1	Between Solvers	50
3.3.2	Between Programs	51
3.3.3	Between Instances	52
3.4	Structure Generalisation	53
3.4.1	From Solvers	54
3.4.2	From Programs	54
3.4.3	From Instances	54
3.5	Additions to the Framework	55
3.6	Conclusion	56
4	Preserving Structure	57
4.1	Introduction	57
4.2	Background	59
4.2.1	Presolving Linear Programs	59
4.2.2	Presolving Mixed Integer Programs	62
4.2.3	Presolving Boolean Satisfiability	65
4.2.4	Presolving Constraint Programs	66
4.2.5	Related Work in Compilation	67
4.3	Multi-Pass Presolving	69
4.3.1	Multi-Pass Examples	69
4.3.2	Presolving Phases	72
4.3.3	Preserving Structure	75
4.3.4	Variable Paths	76
4.4	Implementation in MiniZinc	77
4.4.1	Compilation in MiniZinc	77
4.4.2	Implementing Variable Paths	77
4.5	Experimental Evaluation	80
4.5.1	Two Pass Compilation: For MIP	81
4.5.2	Two Pass Compilation: For CP	85
4.5.3	Two Pass Compilation: Case Study	88
4.6	Conclusion	93
4.6.1	Limitations	93
4.6.2	Further Research	93

5	Discovering Structure	95
5.1	Introduction	95
5.2	The Progressive Party Problem	97
5.3	Related Work	100
5.4	Globalization	102
5.4.1	Normalization	103
5.4.2	Generating Submodels	104
5.4.3	Instantiating Submodels	106
5.4.4	Processing Submodel Instance Groups	107
5.4.5	Preprocessing Step to Uncover Hidden Structures	110
5.5	Implementation	113
5.5.1	Implementation Choices	114
5.5.2	Library of Global Constraints	114
5.5.3	Displaying Candidates	115
5.6	Experimental Evaluation	116
5.6.1	Evaluating the Default Configuration	116
5.6.2	Evaluating Alternative Configurations	119
5.7	Conclusion	122
5.7.1	Limitations	123
5.7.2	Further Research	123
6	Structure Guided Fault Diagnosis	125
6.1	Introduction	125
6.2	The Latin Squares Problem	127
6.3	Related Work	130
6.4	Exploiting Model Structure for MUS Detection	133
6.4.1	Constraint Paths	133
6.4.2	Grouping Constraints by Paths	136
6.4.3	Implementation Details	140
6.5	Experimental Evaluation	142
6.5.1	MUS enumeration	144
6.5.2	Time to First MUS	145
6.6	Displaying Diagnoses	146
6.7	Generalising MUSes to the Model Level	148
6.8	Conclusion	152
6.8.1	Limitations.	152
6.8.2	Further Research.	153
7	Conclusions	154
	Appendix A Preserving Strucutre: Models	159

Appendix B Discovering Structure: Models	160
B.1 Cars	160
B.2 Jobshop	160
B.3 Party	160
B.4 Party-CSP	161
B.5 Party-LP	161
B.6 Packing	162
B.7 Schedule	163
B.8 Sudoku LP	163
B.9 Sudoku CSP	164
B.10 Warehouses	165
Appendix C Structure Guided Fault Diagnosis: Models	166
C.1 Costas-Array	166
C.2 CVRP	167
C.3 RCMSP	170
C.4 Free-Pizza	170
C.5 Mapping	171
C.6 MKnapsack	173
C.7 NMSeq	174
C.8 Open-Stacks	174
C.9 P1F	175
C.10 Radiation	177
C.11 Spot5	178
C.12 TDTSP	179
Vita	181

List of Tables

4.1	Two pass compilation for a MIP solver experiment.	82
4.2	Per-model summaries for MIP experiment.	84
4.3	Two pass compilation for a CP solver experiment.	85
4.4	Per-model summaries for CP experiment.	87
4.5	Path-based two pass presolving comparison for the RCMSP problem.	91
5.1	Library of global constraints supported by Globalizer.	115
5.2	Globalizer experiment.	119
5.3	Comparison of different configurations.	120
6.1	Enumerating multiple diagnoses.	144
6.2	Finding the first diagnosis.	145

List of Figures

1.1	Example solution to an instance of the Car Sequencing Problem.	2
1.2	Traditional modelling, compilation, and solving process.	4
1.3	A slide from Régis’s 2013 Research Excellence Award presentation.	8
1.4	MiniZinc IDE showing combination of constraints that cause failure.	11
2.1	MIP formulation of the Golomb Ruler Problem.	19
2.2	SAT formulation of the Golomb Ruler Problem.	21
2.3	Constraint Model of the Golomb Ruler problem.	26
2.4	MiniZinc model and two data files for the Golomb Ruler problem.	27
3.1	New framework showing structure generalisation and structure sharing. . .	49
3.2	Trace of compilation of Golomb Ruler model.	52
3.3	New framework showing new additions.	55
4.1	Example configuration of the multi-pass presolving approach.	74
4.2	Preserving explicit structure to the program-level.	75
4.3	Trace of compilation for introduced variables.	76
4.4	Problems selected for use in evaluation of multi-pass approach.	82
4.5	Comparison of single and two pass compilation for a MIP solver.	83
4.6	Comparison of single pass and two pass compilation for a CP solver.	86
4.7	Geometric mean optimality gap for the RCMSP problem.	90
4.8	Total solve time for the RCMSP problem.	92
5.1	Graphical overview of model globalization.	102
5.2	The web interface to the MiniZinc Globalizer.	116
5.3	Experiment looking at different configurations.	121
6.1	A reduced trace tree for the Latin Squares model.	135
6.2	Selective Deepening.	139
6.3	Incomplete Deepening.	141
6.4	Incomplete enumeration can miss some MUSEs.	142
6.5	Experiment configurations.	143
6.6	Over-constrained Latin Squares in the MiniZinc IDE.	147
6.7	Prototype MiniZinc IDE showing different diagnoses.	148
6.8	Full trace tree for RCMSP instance.	151
6.9	Generalised tree for the RCMSP problem.	151

List of Algorithms

2.1	The mzn2fzn driver.	33
2.2	Flattening an instance.	34
2.3	Flattening a variable declaration.	35
2.4	Adding new variables.	35
2.5	Adding constraints.	35
2.6	Helper procedures for flattening expressions and arrays.	36
2.7	Flattening a constraint.	37
2.8	Flattening a term.	38
2.9	Flattening a let expression.	39
4.1	Integrating variable paths into compilation.	79
4.2	Procedure that returns a path uniquely identifying this compilation unit.	80
4.3	getCurrentPath with optimisations.	81
5.1	High-level algorithm for Globalizer.	102
5.2	Flatten conjunctions and aggregate forall quantifier to normalize.	103
5.3	Splitting a set of constraints into submodels.	104
5.4	Instantiating a set of submodels.	106
5.5	Processing submodel instance groups.	107
5.6	Generating candidate constraints for a submodel instance.	108
5.7	Ranking submodel instances.	110
6.1	Adapted MARCO algorithm.	132
6.2	Satisfiability check.	132
6.3	Annotating program constraints with paths.	134
6.4	Function that groups constraints that share a prefix.	137
6.5	Procedure for reporting MUSes found at different depths.	137
6.6	Mapping labels to deeper constraint groups.	138

Making the Most of Structure in Constraint Models

Kevin Leo

Monash University, 2017

Supervisor: Dr. Guido Tack

Associate Supervisor: Prof. Maria Garcia de la Banda

Abstract

Combinatorial problems are those where a set of decisions that satisfy a given set of constraints, must be selected from a set of possible combinations. Solving these problems is critical in a wide range of areas, such as telecommunications, circuit design and transportation. It is also remarkably hard, which is why many solving techniques exist, each with their own strengths and weaknesses. High-level modelling languages allow combinatorial problems to be specified as high-level models that can then be compiled into different low-level programs, each suitable for a solving technique. While the high-level modelling and solver-independence characteristics of these languages are very useful for modellers, they make it difficult for the compiler to yield programs that can be solved efficiently. Further, changes to a model can have a significant and unexpected impact on the efficiency of the resulting programs. This makes the modelling, compilation, and solving of combinatorial problems a challenging iterative process.

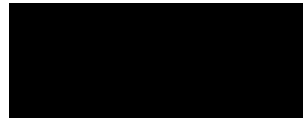
This thesis states that information regarding the implicit and explicit structure present in a model can be very beneficial for this challenging process, as it allows users, compilers and solvers to make better decisions during modelling, compilation, and solving. The main objective of the thesis then, is *to effectively take advantage of structure in constraint models to improve the modelling, compilation, and solving process*. To achieve this, a new framework is proposed for exploring how structural information can be discovered, shared, and generalised between different stages of the process. In particular, the framework is used as a guide to determine whether the explicit structure of a model can be used to (a) improve the quality of compiled programs; (b) guide the search for and exposure of hidden implicit structure; and (c) help identify parts of a model that are incorrectly formulated.

This is achieved by the following three main contributions of this thesis. First, a method that preserves model structure deeper into the compilation process, thus enabling more efficient programs to be produced. Second, a method for finding implicit structure that can be made explicit in a model, thus making modelling easier. And third, a method that uses preserved model structure to help explain why a model is unsatisfiable, thus helping users produce correct models. These three methods have been implemented and evaluated for the MiniZinc language. The experimental results show them to be effective in practice. Together, these contributions help make high-level modelling a more powerful and attractive approach for solving hard combinatorial problems.

Making the Most of Structure in Constraint Models

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.



Kevin Leo
July 11, 2017

Acknowledgments

This work would not have been possible without the support and friendship I received while at Monash. First and foremost, I want to thank my supervisors, Guido and Maria, for everything they have done for me over the past four years. I thank them for providing excellent advice, encouragement, and feedback, and for the long hours they have put into making this thesis possible. I am truly honoured to have had such an excellent team guiding me on this journey.

A debt of thanks is owed to the administrative staff at Caulfield, who have looked after me so well. In particular, I'd like to thank Allison, who has always put in the extra work to make sure that everything ran smoothly.

I'd like to thank my friends and colleagues at Monash University, for stimulating discussions about programming, fitness, literature, and even a little bit of politics and for board-gaming, hiking, rock climbing, frisbeeing, barbecuing, juggling, road tripping, movie watching, and excessive tea drinking. In no particular order, I'd like to thank: Steve, Richard, Maxim, Tom, David, Elizabeth, Peter and Dora for being wonderful friends.

This research was funded in part by a NICTA VRL Enhanced PhD Scholarship. I would like to thank NICTA first for providing financial support and access to the brilliant researchers in the optimisation research group (ORG). This research was also partly sponsored by the Australian Research Council grant DP110102258.

I also want to thank my friends from Cork, Elaine, Padraig, Megan, Barry, and Charles, who have now spread out, but still find time to hang out and play virtual boardgames. I wish to thank my Mom and Dad, Anne-Marie, John, and Yvonne for their continued support from the other side of the world. I especially wish to thank Sandra, for her support, and for keeping me sane in the final months of this work.

Kevin Leo

Monash University
July 2017

Chapter 1

Introduction

Combinatorial problems are those where there is a set of decisions to be made, a set of constraints on these decisions and, in the case of combinatorial optimisation problems, an objective function that measures the quality of a solution. Solving a combinatorial problem requires finding a solution for it, that is, finding a set of decisions that satisfy the set of constraints and optimise the objective function (i.e., minimise or maximise), if any. Combinatorial satisfaction and optimisation problems occur in virtually every organisation, and solving them is critical in a wide range of diverse fields such as telecommunications, emergency management, circuit design, bioinformatics, energy management, transportation, network management and configuration, production scheduling and drug design (Freuder, 1997).

An example of a real-world combinatorial problem is that of the car sequencing problem (Bergen et al., 2001; Dincbas et al., 1988), where a set of cars must be assembled at a factory. The factory can install different optional components (air-conditioning, sun-roof, etc.) on the cars, by using an assembly station for each option. These stations are limited in the number of cars they can handle within a certain time period. For example, a station may only be capable of installing two sun-roofs in any three time slots. The goal then is, given a number of cars to assemble along with the options that are required for each, produce a solution schedule for their assembly, i.e., a schedule that ensures they can all be constructed without overloading any of the assembly stations.

Let us focus on a special form of this problem that asks the question: Is it possible to find a schedule where there are only as many time slots as there are cars to be constructed? Figure 1.1 shows a particular instance of this problem with five assembly stations (Sun Roof, Air-con, etc.), each with their associated limits represented on the right as $N : M$ (N cars within a period of M time slots), and with six classes of car (1 to 6), where the assembly stations required by each car are represented by a dot within the appropriate circle. A possible schedule is demonstrated by the sequence of cars shown at the bottom of the figure. This schedule is a solution, as the constraints on the number of cars each station can handle within a certain period are satisfied. For example, sun roofs can only be installed once in every five time slots (1:5). There are a total of ten cars to be scheduled here and, thus, only ten time slots available. This means that only two cars that require a

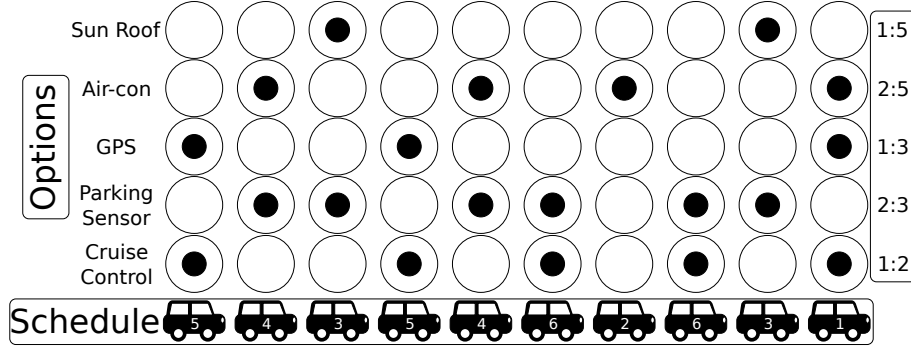


Figure 1.1: Example solution to an instance of the Car Sequencing Problem.

sun roof can possibly be processed in the schedule. The sun roof is only required by cars of class 3, of which only two are required. Therefore, this constraint can be satisfied.

Combinatorial problems are remarkably difficult to solve efficiently, due to the exponential number of possible combinations of decisions that need to be explored when searching for a solution. There has been a significant amount of research in developing and evaluating a wide range of general purpose solving techniques, some of which can speed up the search for a solution by orders of magnitude. When packaged as stand alone programs, these techniques are called *solvers*. Solvers take a representation of a problem as input and either output a solution or, in the case that there is none, report *unsatisfiability*.

High-level modelling languages for combinatorial problems such as Zinc (Garcia de la Banda et al., 2006), ESSENCE (Frisch et al., 2007), OPL (Van Hentenryck et al., 1999), and MiniZinc (Nethercote et al., 2007) allow combinatorial problems to be described in a *model* in a solver-independent way, that is, to be modelled in a high-level language and then by translated for solving by one of a wide selection of solvers. This is useful, as no solver is consistently better for all combinatorial problems and high-level models are often much clearer than low-level ones. However, achieving solver-independence is not straightforward, since not all solvers can understand the same input formats, as they approach solving in different ways. In this thesis we differentiate between constraint models and *programs*. Models are compiled from their high-level representations into low-level *programs* which target specific solvers. Importantly, the particular structures used to model a given problem will have a major impact on the efficiency of the resulting programs. For example, modelling a problem in terms of Boolean (zero-one) decisions may make the search for solutions harder for certain solvers, when compared to modelling the problem in terms of integer decisions. This is, among other reasons, because the knowledge provided by these structures can help the compiler and solver make good decisions when compiling and solving the model. Thus, developing a good model, that is, one that results in correct and efficient programs, is critical for program efficiency.

Unfortunately, while developing naive models can be relatively easy, developing good models is often a challenging iterative process that requires considerable levels of expertise and consumes significant resources. The aim of this thesis is to improve this situation, that is, reduce the level of expertise and the amount of resources needed to develop good

models, by making better use of the structures present in the model. In particular, the main objective of the thesis is:

To effectively take advantage of structure in constraint models to improve the modelling, compilation, and solving process.

To achieve this objective a framework is proposed that can help us reason about the flow of knowledge and structural information within the modelling, compilation, and solving process. Using this framework as a guide, the thesis presents three approaches that use structure from constraint models to improve this process. The first approach develops and uses a representation of the *explicit* structure of a model to improve the compilation of models for target solvers. By explicit structure we refer to any information syntactically present in the model, from the type of variables and the name of the constraints used, to annotations marking a constraint as redundant and a set of variables as symmetric. The second approach utilises explicit model structure to guide the search for *implicit* structure. By implicit structure we refer to any model information that is not explicit, but is implied by the explicit structure. We are interested in implicit structure that, if made explicit in the model, could help modellers, compilers, and solvers make better decisions. Finally, the third approach extends the representation of explicit model structure developed before, and uses it both to quickly diagnose modelling mistakes that cause unsatisfiability, and to explain them to the user in an understandable way.

1.1 The Modelling, Compilation, and Solving Process

Combinatorial problems are typically specified in terms of a set of *parameters* representing the input data, a set of *variables* representing the decisions that must be made, a set of *domains* representing the possible options for each variable, and a set of *constraints* representing the relationships between variables. In the case of an optimisation problem, an *objective function* must also be specified, which measures the quality of a solution.

High-level modelling languages allow models of combinatorial problems to be solved using a wide selection of *solvers*. There are several types of solvers available for combinatorial problems, most notably Constraint Programming (CP), Boolean Satisfiability (SAT), Mixed-Integer Programming (MIP), Mixed Integer Non-Linear Programming (MINLP) and Satisfiability Modulo Theory (SMT) solvers. Different types of solver require problems to be described in different ways. For example, CP solvers solve problems described in terms of complex non-linear *global constraints* with non-linear objectives (see Section 2.3.4), while MIP solvers require problems to be formulated in terms of systems of linear equations with a linear objective function (see Section 2.3.2), and SAT solvers require problems to be described in terms of logical formulae over Boolean variables (see Section 2.3.3). Since no solver dominates in performance for all problem classes, it is important for modellers to be able to easily experiment with different solvers to find what works best for their specific problem. This is achieved by the modelling, compilation, and solving process shown in Figure 1.2. The modeller starts the process by formalising their problem

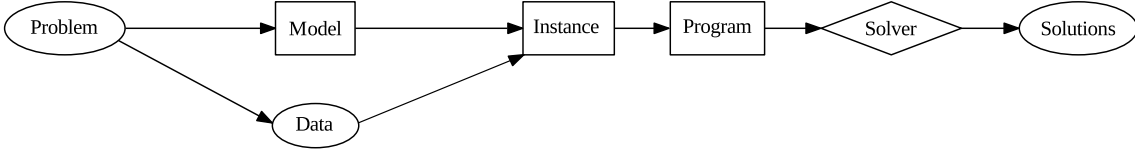


Figure 1.2: Traditional modelling, compilation, and solving process.

as a high-level parametric model. Some input data for the parameters of the model must also be provided, often in the form of a separate file.

Let us illustrate this with an example in the popular MiniZinc (Nethercote et al., 2007) modelling language. MiniZinc was chosen for this thesis for several reasons. First, it is a solver independent modelling language with support for many CP and MIP solvers. Please note that as a result, this thesis focusses mainly on CP and MIP solvers. This makes experimentation with different technologies very easy. Second, The language itself has features such as user-defined predicates and functions which allows more problem structure to be expressed explicitly in the model. Third, The popularity and maturity of the language has resulted in a relatively large suite of benchmark models being available, many of which have come from the MiniZinc challenge (Stuckey et al., 2010). And fourth, the MiniZinc language now has a standard IDE (The MiniZinc IDE) where the tools presented in this thesis can be integrated. In addition, access to MiniZinc expertise was on hand, since the new MiniZinc compiler (Tack, 2011) was in development at Monash University. The compiler has several features that were required and, since it was still in-development, new features could be added more easily. Note that, despite the use of MiniZinc, the methods explored in the thesis should be applicable to other languages. Listing 1.1 and Listing 1.2 present a MiniZinc model and data file for the Car Sequencing Problem described above. It is not necessary to fully understand this example, it is here simply to illustrate the concepts that will be explored later in the thesis.

The model starts with a comment (marked by the symbol `%`), which provides the name of the file (`cars.mzn`). It then includes the library file `globals.mzn`, which will be described later. The model continues by declaring the parameters of the problem in lines 4-13. Keywords of the language are shown in bold to make reading the model easier. The first three parameters declared are `n_cars`, `n_options` and `n_classes`, which will hold the number of cars, the number of options and the number of classes of cars, respectively. Next, the model uses these three parameters to define three sets that will be used to make the remaining declarations more readable. Next, it declares four array parameters: `max_per_block`, `block_size`, `cars_in_class` and `need`. These represent, respectively, the number of cars that a station can process within its period, the period of each station, the number of cars of each class required, and what options each class requires. After the parameters are defined, the model declares its decision variables in lines 16-19. In particular, it first defines the main array of variables `class`, which will contain the solution schedule (if any), that is, the class that should be constructed at each timeslot. The other array, `used`, defines intermediate variables that make it easier to express the constraints of the model. These are defined in lines 23-33. The first constraint, on line 23 connects


```

1  % cars.mzn
2  include "globals.mzn";
3
4  int: n_cars; int: n_options; int: n_classes;
5
6  set of int: steps = 1..n_cars;
7  set of int: options = 1..n_options;
8  set of int: classes = 1..n_classes;
9
10 array [options] of int: max_per_block;
11 array [options] of int: block_size;
12 array [classes] of int: cars_in_class;
13 array [classes, options] of 0..1: need;
14
15 % The class of car being started at each step.
16 array [steps] of var classes: class;
17
18 % Which options are required by the car started at each step.
19 array [steps, options] of var 0..1: used;
20
21 % Block p must be used at step s if the class of the car to be
22 % produced at step s needs it.
23 constraint forall (s in steps, p in options) (used[s, p]=need[class[s], p]);
24
25 % For each option p with block size b and maximum limit m, no consecutive
26 % sequence of b cars contains more than m that require option p.
27 constraint
28     forall (p in options, i in 1..(n_cars - (block_size[p] - 1))) (
29         sum (j in 0..(block_size[p] - 1)) (used[i + j, p])
30         <= max_per_block[p]);
31
32 % Require that the right number of cars in each class are produced.
33 constraint forall (c in classes) (count(class, c, cars_in_class[c]));
34
35 solve satisfy; % Find any solution.

```

Listing 1.1: Model for the Car Sequencing Problem (cars.mzn).

the `class` variables which select the class to use at a certain step, and the `used` variables which record the active stations at a certain step. This is achieved using what is called an *element constraint*, where a variable (`class[s]`) is used to select from an array of values. The next set of constraints starting on line 30 restrict the number of cars that can be processed within each station's period. Finally, the constraint on line 33 maintains that the correct number of cars in each class must be constructed. It does this by calling the *count constraint* which says that for each class `c`, the number of occurrences of this class in the array `class` must be equal to `cars_in_class[c]`. The last item in the model, the *solve* item, states that the goal is to find any solution that satisfies the constraints.

High-level modelling is not only important because it allows problems to be described in a natural way (using high-level rather than solver specific formulations), but also because it allows flexibility during compilation. One powerful contribution of CP is the idea of encapsulating subproblems that crop up frequently as single *global* constraints. An example of a global constraint is that of the *count* constraint included in the model presented above. This constraint could be represented using a set of constraints that record whether or not a value occurs in each position of an array, and then a linear sum constraint over

```

% cars1.dzn
n_cars      = 10;
n_options   = 5;
n_classes   = 6;
max_per_block = [1, 2, 1, 2, 1];
block_size   = [2, 3, 3, 5, 5];
cars_in_class = [1, 1, 2, 2, 2, 2];

need = [| 1, 0, 1, 1, 0 |
        0, 0, 0, 1, 0 |
        0, 1, 0, 0, 1 |
        0, 1, 0, 1, 0 |
        1, 0, 1, 0, 0 |
        1, 1, 0, 0, 0 |];

```

Listing 1.2: Example data file for cars.mzn.

```

int: n_cars = 10; int: n_options = 5; int: n_classes = 6;

set of int: steps = 1..n_cars; % 1..6
set of int: options = 1..n_options; % 1..5
set of int: classes = 1..n_classes; % 1..6

array [1..5] of int: max_per_block = [1, 2, 1, 2, 1];
array [1..5] of int: block_size = [2, 3, 3, 5, 5];
array [1..6] of int: cars_in_class = [1, 1, 2, 2, 2, 2];
array [1..6, 1..5] of 0..1: need = [| 1, 0, 1, 1, 0 |
                                         0, 0, 0, 1, 0 |
                                         0, 1, 0, 0, 1 |
                                         0, 1, 0, 1, 0 |
                                         1, 0, 1, 0, 0 |
                                         1, 1, 0, 0, 0 |];

. . .

```

Listing 1.3: Instance of cars.mzn model instantiated with parameters from Listing 1.2.

these stating that they must sum to a particular value. By stating the constraint as a **count** constraint instead, the model is not just easier to read, but also allows the compiler more flexibility when it comes to compilation. Depending on the type of solver being targeted, the compiler may present the constraints as they are (i.e., as globals) or decompose them into simpler constraints if the solver does not possess a dedicated algorithm for handling **count** constraints.

Once a model is defined, the compiler combines it with given input data to form an *instance* level representation of the model, that is, a non-parametric model of the problem instance. Listing 1.3 shows an instantiation of Listing 1.1 where the parameters have been bound to the values of Listing 1.2. Note that the exact dimensions of the arrays and matrix are now known.

An instance is then translated by the compiler into a *program*, which can be understood and solved by a target solver. This is usually achieved by unrolling all loops and decomposing constraints that cannot be handled by the target solver into constraints that describe the same relationship in a way that the solver can handle. Note that a single instance can be compiled into different programs, depending on the target solver. Listing 1.4 shows a small extract from the result of compiling the instance above for a CP solver that

supports both linear constraints and the `count` constraints. The program contains some variables with an `i` prefix, indicating integer variables *introduced* by the compiler. The program also contains linear constraints, `int_lin_le`, which take as arguments an array of coefficients, an array of variables, and a bound. The first constraint listed corresponds to linear constraint $i_{84} + i_{89} + i_{94} + i_{99} + i_{104} \leq 1$. The program also shows some of the `count` constraints from the model, which restrict the number of times a certain class can occur in the array `class`.

Note that much of the explicit structure present in the original model, such as the way in which constraints and variables are grouped, has been so obscured as to be practically lost, making it extremely challenging to recover. As we will see later, this has repercussions regarding the kinds of compiler and solver optimisations that can be performed.

1.2 The Importance of Good Models

During his acceptance speech for the 2013 Research Excellence Award by the Association of Constraint Programming, Jean-Charles Régin reinforced my view of the importance of modelling (and the models themselves) when solving combinatorial problems (Régin, 2013). Figure 1.3 shows one of Régin’s slides where he argues that, while model improvements are difficult, they are still worth attempting as the reward in performance gained can be considerable. Despite the fact that these numbers are merely anecdotal, they resonate with what I have observed to be the case in practice.

Intuitively, the approach for improving models taken in this thesis is to analyse them in order to detect, represent, and communicate structures, both implicit and explicit, that can be used to improve modelling, compilation, and solving.

Research Objective

Currently, there is a gap in the knowledge related to the use of structure to support the modelling, compilation, and solving process. Therefore, the primary research objective that this thesis seeks to satisfy is *to effectively take advantage of structure in constraint models to improve the modelling, compilation, and solving process*.

As the basis for this work a new framework for reasoning about how models are compiled is presented. It shows the many possible ways that information can be shared and transferred within the modelling, compilation, and solving process. This work is of significance, as it provides a framework for the study of structure usage in constraint models.

```

153 constraint int_lin_le([1,1,1,1,1],[i84,i89,i94,i99,i104],1);
154 constraint int_lin_le([1,1,1,1,1],[i89,i94,i99,i104,i109],1);
155 constraint int_lin_le([1,1,1,1,1],[i94,i99,i104,i109,i114],1);
156 constraint count(class,1,1);
157 constraint count(class,2,1);
158 constraint count(class,3,2);

```

Listing 1.4: Extract from program compiled from cars.mzn.

General considerations

- **When solving a problem in CP:**
 - Potential performance gain:
 - ▢ data structure optimization (code): x 10
 - ▢ search strategies: x 1 000
 - ▢ model : x 1 000 000
 - Chance of success
 - ▢ data structure optimization (code): 95 %
 - ▢ search strategies: 1 %
 - ▢ model: 0,001 %
 - In this talk, I will mainly speak of modeling

JC Régim - CP - 2013

Figure 1.3: A slide from Régim’s 2013 Research Excellence Award presentation.

The framework is presented in Chapter 3. With this in mind, the search for answers to the following three questions are tackled as sub-goals:

1. Can the explicit structure in models be used to improve the quality of programs compiled from them?
2. Can the explicit structure of a model guide the search for and exposure of implicit structure?
3. Can the explicit structure of a model be used to help identify parts of the model that are incorrectly formulated?

Can the explicit structure in models be used to improve the quality of programs compiled from them?

During compilation, the compiler must make decisions about how different constraints will be compiled. For example, constraint $x \neq y \rightarrow y \leq z$ might need to be broken down into the three constraints $b_1 \leftrightarrow x \neq y$, $b_2 \leftrightarrow y \leq z$, $b_1 \rightarrow b_2$. If the compiler knew that b_1 was going to be fixed to be true, due to constraints processed later in the compilation, it could have compiled these constraints more concisely as just $y \leq z$. One way to allow the compiler to have access to this information is to compile the model twice, first for a solver that operates on a higher-level representation that can find good tighter domains for the variables in a problem (e.g., b_1 being true), and then for the solver that the user intends to solve the problem with, which may use a different representation. In a second compilation, more information (e.g. domains) is made available to the compiler, allowing it to make better decisions.

Automatic approaches for sharing variable domains between different solvers typically focus on sharing between variables that have the same name. This usually means only

sharing domains for the variables that occur in the model. Unfortunately, compilation is a complex process during which many intermediate variables (such as the b_i variables above) are introduced, as they are required for some constraints to be represented in a form supported by the target solver. Compiling an instance for different solvers reveals that the names of these intermediate variables can end up being quite different, making the task of communicating information inferred in the first pass to the second much harder.

To complete this sub-goal, the thesis presents a method that *preserves the structure of a constraint model* deeper into the compilation process, allowing these intermediate variables to be matched across compilations. To achieve this, the compiler is instructed to record a trace starting from the model and continued through compilation, all the way to the point at which each specific variable is introduced in the output program. This provides a unique identifier for variables (referred to as a *variable path*) that is resilient to compilation, even when parts of the model are compiled in a different order.

Consider, for example, the compilation of the Car Sequencing model presented in Listing 1.1 for a solver that does not support the `count` global constraint directly. The constraint will be expressed in a form that the solver can understand by breaking it into simpler constraints. This is possible because we can use the same model with different target solvers, but it also means that the `count` constraint no longer occurs explicitly in the program. The approach taken is to first compile for a solver that does support this constraint, then use the dedicated algorithms of that solver to perform some initial analysis of the instance, and finally use what is learned to produce a better program for the target solver.

The significance of the approach is that a representation of the structure of a constraint model is now able to survive compilation. This allows for greater sharing of information between programs derived from the same instance and, as a result, allows problems to be both compiled and solved faster, making the use of high-level modelling languages more attractive for users.

The approach, presented in Chapter 4, was implemented for MiniZinc and has been shown to improve the performance of compiled programs, particularly when compiling for target MIP solvers, where it was able to reduce the size of the formulations considerably, making compilation faster, and improving solve times. The implementation is part of the MiniZinc compiler and is scheduled for release. This work was published in Leo et al. (2015):

Kevin Leo and Guido Tack, “Multi-Pass High-Level Presolving”. *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pp. 346–352.

Can the explicit structure of a model guide the search for and exposure of implicit structure?

The goal here is to suggest improvements for a model by making use of the explicit structure already present to detect and make implicit structures, such as inverse variables and global constraints, explicit. Making global constraints explicit in a model is in general

advantageous, as they might improve the modeller’s understanding of the problem, allow compilers to make better choices about compilation, and solvers to solve problems faster.

Previously, it has been demonstrated that some global constraints present in a problem instance can be exposed and explicitly stated based on sample solutions to the problem (Beldiceanu et al., 2012). However, this approach only finds global constraints for a single instance. To achieve this goal, the thesis goes beyond the instance level and seeks to discover globals at the model level by using an already existing model to guide the search.

The approach, presented in Chapter 5, is to first split models into many possible submodels. For each submodel, a set of candidate global constraints are constructed using the explicit structure present in them. Finally, sample solutions for each submodel and candidate global constraints for various instances, are generated and tested by checking that a) all sample solutions of one can be accepted as solutions to the other and b) that the candidate global constraints exist in all previously explored instances. The experimental evaluation of this approach has shown that, for many models, we can correctly detect global constraints that were not explicitly stated in the model. While the approach does not prove the correctness of adding these global constraints, it is expected that users will not find it difficult to decide whether they are appropriate.

Consider again the `cars.mzn` model presented above. This model has several constraints that could be expressed more succinctly using global constraints. Using the approach presented in Chapter 5 these constraints can be found. Two global constraints in particular can be added to strengthen the model. The first is a `sliding-sum` constraint that can replace part of the constraint on line 30 of `cars.mzn`. The `sliding-sum` constraint maintains that the sum of any contiguous subsequence of variables of a certain length in an array is between some bounds. The constraint is defined by four parameters: a lower bound for the sum of each subsequence, an upper bound, the length of the subsequences, and the array itself. The constraint can be added as follows:

```

25 % For each option p with block size b and maximum limit m, no consecutive
26 % sequence of b cars contains more than m that require option p.
27 constraint forall(p in options) (
28     sliding_sum(0, max_per_block[p],
29                block_size[p],
30                [used[i, p] | i in 1..n_cars]));

```

This constraint is now a single `forall` loop over each `p` in `options`. For each option, we introduce a `sliding-sum` constraint with a lower bound of 0, an upper bound defined by the value `max_per_block[p]`, the length set by the value `block_size[p]`, and finally an array of `used` variables corresponding to the selected option at each time slot is constructed. The second constraint that can be added is the `global-cardinality` constraint. This subsumes the entire group of `count` constraints introduced on line 33 resulting in the following:

```

32 %% Require that the right number of cars in each class are produced.
33 constraint global_cardinality(class,
34                               [i | i in 1..n_cars]),
35                               cars_in_class);

```

```

File Edit MiniZinc View Help
Configuration cars.mzn X
16 array [steps] of var classes: class;
17
18 % Which options are required by the car started at each step.
19 array [steps, options] of var 0..1: used;
20
21 % Block p must be used at step s if the class of the car to be
22 % produced at step s needs it.
23 constraint forall (s in steps, p in options) (used[s, p]=need[class[s], p]);
24
25 % For each option p with block size b and maximum limit m, no consecutive
26 % sequence of b cars contains more than m that require option p.
27 constraint
28   forall (p in options, i in 1..(n_cars - (block_size[p] - 1))) (
29     sum (j in 0..(block_size[p] - 1)) (used[i + j, p])
30     < max_per_block[p]);
31
32 % Require that the right number of cars in each class are produced.
33 constraint forall (c in classes) (count(class, c, cars_in_class[c]));
34
Output
Conflict:9: s=2, p=5; s=1, p=4; s=2, p=4; s=1, p=1; p=1, i=1; s=2, p=1; p=5,
i=1; p=4, i=1; s=1, p=5
Conflict:9: s=9, p=4; p=5, i=6; s=10, p=4; s=9, p=1; s=9, p=5; p=1, i=9; s=10,
n=5: n=4. i=6: s=10. n=1
Ready.

```

Figure 1.4: MiniZinc IDE showing combination of constraints that cause failure.

This constraint takes three parameters: an array, a cover, and a set of counts. The cover defines the values that must occur a certain number of times in the array, while the counts define these numbers. In this case, the array is `class`, which holds variables representing the manufacturing schedule. The cover is defined as an array of the numbers 1 to `n_cars`. The counts are then presented by the parameter array `cars_in_class`. Replacing the constraints in the model with these higher-level global constraints can make the model easier to read, allow compilers to make better decisions about how to compile an instance, and allow instances for a CP solver that has dedicated algorithms for these constraints, to be solved faster.

This work was implemented for MiniZinc and was made available as an online tool called MiniZinc Globalizer. Experimental evaluation has shown that this is a good approach for exposing implicit structure in constraint models. While the online service is no longer available, a stand alone version that integrates with the MiniZinc IDE is planned. This will allow users to more easily integrate this tool into their existing workflow without having to copy and paste their model to an online service. This work was published as Leo et al. (2013):

Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda,
 “Globalizing Constraint Models”. *Principles and Practice of Constraint Programming, CP 2013, Proceedings*, pp. 432–447.

Can the explicit structure of a model be used to help identify parts of the model that are incorrectly formulated?

Modelling combinatorial problems can be hard initially. In fact, a first attempt at modelling a problem will often result in an unsatisfiable model, that is, one in which no solution that satisfies the constraints of the model can be found. Having an explanation of what

constraints in a model have lead to the unsatisfiability can be useful for helping users discover and correct their mistakes. Existing approaches focus on finding explanations in terms of compiled programs. A common approach is to search for what are called minimal unsatisfiable sets, or MUSes (Bailey et al., 2005; Junker, 2001; Liffiton et al., 2013). These MUSes try to succinctly represent the cause of unsatisfiability.

There are two main limitations of current MUS-based diagnosis approaches. The first is that MUSes are found in the set of program-level constraints and, thus, it may be difficult for a user to map them back to the model they wrote. The second limitation is that finding MUSes in large programs can be prohibitively slow, as MUS detection tries to explore the power-set (all combinations) of constraints in the program.

The approach taken here extends the concept of variable paths to *constraint paths*, which identify the path from a model through compilation to a specific constraint in the program. Constraint paths *make the hierarchy of constraints in a model explicit, so that they can then be used to intelligently break the search for MUSes into more manageable chunks*. Importantly, the hierarchy also allows *program level constraints to be mapped back to the user model, making it easier for a user to interpret the MUS*.

Figure 1.4 shows an unsatisfiable version of `cars.mzn` where instead of using `<=` in line 30, the modeller wrote `<`. Using the approach presented in Chapter 6 we can highlight in the model which constraints were involved in the failure. In this case, the constraint connecting the `used` and `need` variables, combined with the faulty constraint on line 30, are highlighted. In addition, the interface also presents to a user the specific assignment of index variables in loops that are involved in the fault. The highlighting allows the user to focus on the parts of the model that are causing issues.

A prototype implementation for finding MUSes in MiniZinc models was developed as part of this research by modifying an existing MUS enumeration tool. Integration with the MiniZinc compiler is planned. This work was published as Leo et al. (2017):

Kevin Leo and Guido Tack. “Debugging Unsatisfiable Constraint Models”. *Integration of AI and OR Techniques in Constraint Programming, CPAIOR 2017, Proceedings*, pp. 77–93.

1.3 Contributions

In conclusion, combinatorial problems appear in a wide range of real world problems. High-level modelling makes it feasible to find effective ways to solve these complex problems. This thesis first presents a framework for reasoning about how structural information can be better used and communicated during the modelling, compilation, and solving process. This framework is then used to develop the core contributions of this thesis, which are:

- To design, implement, and evaluate a method that preserves a representation of the structure in a constraint model (variable paths), allowing multiple compilations of the same instance of a problem to be accurately compared and utilised to improve further compilations. This is achieved by introducing the concept of variable paths.

This contribution makes high-level modelling more attractive and efficient, requiring less expertise on the side of the modeller and fewer resources, as it yields smaller more efficient programs.

- To design, implement, and evaluate a method that utilises explicit model structure to guide the search for implicit model structure that can then be made explicit in a later revision of the model. This is achieved by the MiniZinc Globalizer approach. This approach can have a great impact on the quality of models produced and, therefore, its impact can translate to improvements for all instances. This more than makes up for the cost of analysis.
- To design, implement, and evaluate a method that extends the representation of high-level structure (constraint paths) and uses it to aide the search for sets of unsatisfiable constraints that succinctly describe a fault. This presents an improvement in terms of efficiency and allows for more user-friendly presentation of faults. Thanks to this contribution, users will be able to more quickly diagnose and fix issues in their models leading to faster development that requires less specialised expertise.

Together, these methods help modellers produce models that are easier to read, can be solved faster, and have fewer bugs, making high-level modelling more attractive.

1.4 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 explores the necessary background required for understanding the concepts in this thesis. This involves an exploration of combinatorial problems, the related general purpose solving technologies, how the modelling of combinatorial problems is achieved, and how structure is currently used in existing modelling and solving tools. The compilation of MiniZinc models is also explored. Chapter 3 presents the new framework for the modelling, compilation, and solving of combinatorial problems. The techniques explored in later chapters all relate to aspects of this framework, as it provides a map for explaining how information (e.g., structures) flows throughout the process. Chapter 4 introduces the concept of variable paths and how these have been implemented. This is then used for multi-pass presolving, a method that uses the explicit structure in models to help improve the quality of programs compiled from them, and thus, improve solve times. Chapter 5 explores an approach that seeks to use the explicit structure of a model to guide the search for and exposure of hidden implicit structure. This chapter also makes use of a generalisation methodology that allows information learned from instances to be expressed in terms of the model. Chapter 6 extends variable paths into constraint paths and presents an approach that uses this explicit model structure to help identify parts of the model that are incorrectly formulated. The explicit structure of the model is also used in presenting these incorrect parts to the modeller in terms of the model constraints. A case study using another generalisation methodology is also presented. Finally, Chapter 7 summarises and concludes the thesis.

Chapter 2

Background

2.1 Introduction

This thesis studies the inference and use of structural information to improve the process of modelling, compiling, and solving combinatorial problems supported by the MiniZinc tool-chain. This chapter presents the required background for the rest of the thesis and outlines the boundaries of this research. In doing so, it gives a brief introduction to combinatorial problems, to the general purpose tools available to model, compile, and solve them, and how these tools use structural information.

Structure of the Chapter

Section 2.2 provides an introduction to combinatorial problems. Section 2.3 explores the different approaches for solving combinatorial problems that are used in this thesis. Section 2.4 introduces the MiniZinc modelling language by means of a simple example. Section 2.5 provides the grammar for a reduced subset of the MiniZinc language and describes in detail how a MiniZinc model based on this grammar would be compiled. The grammar and associated compilation will be expanded upon in later chapters, as needed. Section 2.6 highlights the importance of using model structure during the modelling, compilation, and solving process and, therefore, the advantages of both retaining explicit structure and rediscovering implicit structure. Finally, Section 2.7 presents an overview of the ways in which the structure available at different levels of abstraction is used to improve the modelling, compilation, and solving of combinatorial problems.

2.2 Combinatorial Problems

Combinatorial problems are those where a set of decisions must be made that satisfy the given constraints. A notable property of combinatorial problems is the presence of discrete (integral) decisions. Combinatorial problems appear in a wide range of areas in diverse fields such as telecommunications, circuit design, bioinformatics, transportation, network management and configuration, and production scheduling (Freuder, 1997). In this thesis, we will focus on both combinatorial satisfaction and optimisation problems.

2.3 Solving Combinatorial Problems

There are many ways to approach the solving of combinatorial problems. We could, for example, implement a dedicated algorithm for each problem we come across. While this can result in a focused and efficient solver, it also requires a huge investment of time, expertise, and other resources. To reduce this investment significantly, an off-the-shelf general purpose solver for combinatorial problems can be used. Mature solvers typically have many years of research and development invested in them, which can make them as efficient as dedicated solvers for many classes of problem. The use of a general purpose solver shifts the work from implementing an efficient solving mechanism, to finding the best way to describe the problem for that solver.

However, while generic solvers may solve many kinds of combinatorial problems, similarly to the no-free-lunch theorem (Wolpert et al., 1997), they often work best for some classes of problem and not so well for others. Further, it is often difficult to predict whether a given problem is a good fit for a specific solver. As a result, it is usually a good idea to attempt to have the problem solved by several different solvers, each following an approach aimed at different classes of problem. To achieve this, we would have to learn how best to describe the problem for each new solver, once again having to invest in attaining specialised knowledge.

This section outlines four common kinds of general purpose solvers used to solve combinatorial problems – linear programming, mixed-integer (linear) programming, Boolean satisfiability and constraint programming – and highlights their specification requirements. The next section will show how to improve the solver-specific specification issue by using a high-level modelling language.

2.3.1 Linear Programming

Linear programming (LP) (Dantzig, 1963) is an early problem solving technique, closely related to the solving of combinatorial problems. Linear programs are described as a set of linear constraints over a set of continuous variables. While LP is rich enough to describe many problems, including some that require variables to take integral values, it is possible to solve linear programs in polynomial time (Papadimitriou et al., 1982). Thus LP an unsuitable language for describing more general NP-hard combinatorial problems, as these are not known to have any polynomial time solving algorithm. Nevertheless, it is an important approach in the field and a key component of the mixed-integer programming (MIP) approach, which has proven to be quite successful in practice and will be described in Section 2.3.2.

There are many implementations of LP solvers, as MIP solvers typically implement their own. These include Gurobi (Gurobi Optimization, Inc., 2015), IBM CPLEX (IBM, 2010), Mosek (MOSEK ApS, 2016) and Express (FICO, 2016). In addition, open source solvers are also available, including GLPK (Makhorin, 2001); SoPlex, used in the SCIP solver (Gamrath et al., 2016); and the COIN-OR Linear Programming solver (CLP), used in the CBC solver (COIN-OR, 2016).

Linear programs are typically presented in the following form:

$$\begin{aligned}
& \text{minimize:} && \sum_{j \in C} c_j x_j \\
& \text{subject to:} && bl_i \leq \sum_{j \in C} a_{ij} x_j \leq bu_i && \forall i=1^m \\
& && l_j \leq x_j \leq u_j && j \in N \\
& && x_j \in \mathbb{R} && j \in C
\end{aligned}$$

where N and C represent the set of all variables and continuous variables, respectively (which in this case are the same set, others will be added later); vector c holds the coefficients for variables in the objective function (e.g., vector $[1, 3, 0, 4]$ corresponds to an objective function $x_1 + 3x_2 + 4x_4$); matrix a holds the coefficients for the constraints in the program; each row is bound by values from the vectors bl and bu , which represent the lower and upper bounds, respectively; and vectors l and u represent the lower and upper bounds, respectively, of the variables x in the program. To represent variables or constraints that are only bound in one direction, or are unbounded, we use infinity or negative infinity as the upper or lower bound.

The Simplex Algorithm

The simplex algorithm originally proposed by Dantzig (1963) is the primary method used for solving linear programs. Since then, there has been much work on improving the algorithm and the implementations provided in modern solvers are extremely efficient.

In a broad sense, the algorithm walks the vertices of the polytope described by the constraints of the program. Thus, every step provides an assignment to the variables that satisfies the constraints of the program. At each step the objective value is checked and the algorithm will only move to a vertex that improves the objective. An important property of LP is that once there is no vertex that can be moved to from the current one that improves the objective value, we proven that the current vertex is the global optimum.

While simplex is the most popular algorithm for solving LPs, other approaches do exist and are used in commercial solvers (Barahona et al., 2000; Wright, 2005). The different approaches have different strengths. For example, one algorithm may be slow when solving an LP from scratch but may be faster when resolving an LP with only small changes (e.g., new variables or constraints). The requirement that an LP solver be able to quickly resolve is important for mixed integer programming.

2.3.2 Mixed-Integer Programming

Mixed-integer (linear) programming (MIP) is similar to LP but includes the concept of integer variables. The presence of these variables means that the simplex algorithm cannot be used to find correct solutions as, even if the constraints are linear, there is no guarantee

that the solutions will lie on the vertices of a polytope described by linear constraints. The integer variables introduce non-linearity to the formulation and require a more complex solving method. As a result, this formulation can be used for describing and solving NP-hard combinatorial problems.

MIP is typically solved by solving LPs that describe subproblems of the MIP, using a variant of the *Branch and Bound* approach suggested in Land et al. (1960) and expanded upon in Dakin (1965). Thus, MIP solvers have efficient implementations of the simplex algorithm built-in. Additionally, some MIP solvers maintain an implementation of an interior-point (or barrier) algorithm, which may suit some problems better (Wright, 2005).

Several commercial MIP solvers are listed in Section 2.3.1. Several open source solvers are also available including SCIP (Gamrath et al., 2016), GLPK (Makhorin, 2001), and CBC (COIN-OR, 2016).

MIP formulations are similar to those of LP. The following shows the extended language available for MIPs.

$$\begin{array}{ll}
\text{minimize:} & \sum_{j \in N} c_j x_j \\
\text{subject to:} & bl_i \leq \sum_{j \in N} a_{ij} x_j \leq bu_i \quad \forall_{i=1}^m \\
& l_j \leq x_j \leq u_j \quad j \in N \\
& x_j \in \mathbb{R} \quad j \in C \\
& x_j \in \mathbb{Z} \quad j \in I
\end{array}$$

Here, the main additions are the set I , which represents the set of integer variables in the problem, and the set N which represents the combined set of integer and continuous variables. The rest of the formulation is essentially the same, but adapted to include these integer variables in the objective and constraints.

Branch and Bound

As mentioned before, MIP solvers typically solve mixed-integer programs by repeatedly solving linear programs with a branch and bound style algorithm. The first step of the algorithm is to relax the integrality constraints and find a continuous solution to the formulation using an LP solver (Linderoth et al., 1999). If the integer variables of the program are given integer values in this initial solution, a solution has been found and the algorithm can stop. Otherwise, the algorithm selects from the integer variables using some variable selection heuristic. One simple heuristic is the *most-fractional value* heuristic, which finds the variable that has taken a value that is the furthest from an integer value. In practice, more complex heuristics are used, such as the *pseudo-costs* heuristic, which take into account information about how variables have interacted with the constraints and objective of the program so far.

Once a variable has been selected, the solver creates two subproblems: The first, with the variable bound to be less than or equal to the floor of its assigned value; and the second, where the variable is bound to be greater than the ceiling of its assigned value. If a subproblem is found to be unsatisfiable the solver backtracks to the previous decision, and continues to search the remaining, unexplored subproblems. When a valid solution (one where the integer variables have been assigned integral values) is found, a bound on the objective value can be updated and used to trivially discard remaining unexplored subproblems if the objective found by solving their continuous relaxation is worse. When there are no more unexplored subproblems, the solver has either proven that the last found solution is indeed optimal (with respect to some tolerance) or, if no solution has been found, that there is no solution and the program is unsatisfiable.

As an example, assume that we have just solved the linear relaxation of a MIP and the solution sets an integer variable x to the value 1.3. The solver will create two subproblems: One for $x \leq \lfloor 1.3 \rfloor$ ($x \leq 1.0$), and the other for $x \geq \lceil 1.3 \rceil$ ($x \geq 2.0$). The solver then recurses into the first subproblem and finds the best solution it can. It then compares this to the best solution it can find in the second subproblem, intelligently pruning the search tree when a continuous relaxation cannot find a solution with a better objective value.

Designing programs for MIP solvers presents two main benefits. First, many available MIP solvers are quite mature after many decades of research and development. Second, the language is quite simple and for some problems can be sufficient to clearly describe a problem. However, the simplicity of the language also means that describing more complex problem constraints can be a real challenge requiring significant experience. In addition, while the solvers are powerful, they contain many moving parts that can interact unpredictably. This means it can be difficult to predict how MIP solvers will perform on a particular formulation of a problem. Thus, one must invest the time and energy to design a program, without knowing if it will be fruitful or not.

Creating Mixed-Integer Programs

For linear relationships between variables, one can simply add linear constraints to the formulation. However, for non-linear relationships one will often have to use more complex formulations that require intermediate variables to be introduced. An example of a type of intermediate variable that is commonly used when modelling problems as mixed-integer programs is that of the binary indicator variable. These kinds of variables are introduced to represent the possible assignments to an integer variable. For example, for an integer variable $0 \leq x \leq 5$, we would introduce six binary variables b_0, \dots, b_5 to represent $x = i \leftrightarrow b_i = 1$. Non-linear constraints over integer variables can then be represented in terms of linear constraints on these binary variables. This approach will be demonstrated below. Another use of binary indicator variables is to allow the degree to which a constraint is satisfied to be measured. For example, the satisfaction of the constraint $x + y + z \leq 10$ can be measured by transforming it into $x + y + z - Mb \leq 10$, where M is selected so that $b = 1$ if $x + y + z$ is not less than or equal to 10. The b variable can then be used in other constraints to represent the truth value of the expression, and can be used in

$$\begin{aligned}
 \text{given: } m & \qquad \qquad \qquad \text{integer} & (2.1) \\
 \text{minimize: } mark_m & & (2.2) \\
 \text{subject to: } mark_1 = 0 & & (2.3) \\
 mark_j - mark_i = d_{ij} & \qquad \qquad \qquad \forall_{i=1}^m \forall_{j=i+1}^m & (2.4) \\
 mark_i - mark_{i+1} \leq 0 & \qquad \qquad \qquad \forall_{i=1}^{m-1} & (2.5) \\
 \sum_{k=0}^{m^2} kb_{ijk} = d_{ij} & \qquad \qquad \qquad \forall_{i=1}^m \forall_{j=i+1}^m & (2.6) \\
 \sum_{i=1}^m \sum_{j=i+1}^m b_{ijk} \leq 1 & \qquad \qquad \qquad \forall_{k=0}^{m^2} & (2.7) \\
 0 \leq mark_i \leq m^2 & \qquad \qquad \qquad \forall_{i=1}^m & (2.8) \\
 0 \leq d_{ij} \leq m^2 & \qquad \qquad \qquad \forall_{i=1}^m \forall_{j=i+1}^m & (2.9) \\
 b_{ijk} \in \{0, 1\} & \qquad \qquad \qquad \forall_{i=1}^m \forall_{j=i+1}^m \forall_{k=0}^{m^2} & (2.10)
 \end{aligned}$$

Figure 2.1: MIP formulation of the Golomb Ruler Problem.

the objective function to penalise satisfaction or non-satisfaction of the constraint. This approach is called a “big-M” formulation, and allows for much more complex problems to be described using MIP.

Mixed-integer programming example. Consider the Golomb Ruler problem of size m , which aims to find a ruler with m marks, where the distances between any pair of marks are distinct and the length of the ruler is minimal. There have been many real-world applications for Golomb Rulers, including carrier frequency assignment (Fang et al., 1977), antenna placement (Thompson et al., 2001), and error correction (Robinson et al., 1967). In Figure 2.1 a MIP formulation of the Golomb Ruler problem is presented. To formulate this problem we first introduce the decision variables $mark_{i..m}$ to represent the positions of the m marks. The objective is to minimise the position of the last mark $mark_m$. Constraint 2.3 states that $mark_1$ must be fixed to 0, as any valid set of marks that do not start at 0 can always be shifted to the left to give a better objective value. Any such non-zero starting solution is said to be *dominated*. Adding this constraint stops the solver from having to explore these dominated solutions. Constraint 2.4 sets the values of a set of introduced variables d to represent the distances between all distinct pairs of marks on the ruler. Constraint 2.5 ensures the marks are ordered as increasing, removing symmetries. Constraint 2.6 establishes the channelling between the introduced variables d and a set of binary indicator variables b to maintain the relationship $d_{ij} = k \rightarrow b_{ijk} = 1$. Constraint 2.7 states that for each distance, at most one of the binary variables representing a d variable can be assigned that distance. This restricts the d variables to be distinct since the distances between the marks on the ruler must take distinct values. This formulation of the well known distinct or `alldifferent` constraint (discussed later when presenting the CP formulation), is covered in Williams et al. (2001). The remainder of the formulation sets

bounds for the marks and introduced variables. For $m = 4$, this formulation results in a program containing a few hundred variables and tens of constraints.

2.3.3 Boolean Satisfiability

Boolean satisfiability (SAT) is a classic problem of computer science (Cook, 1971; Levin, 1973). It poses the question of whether a given Boolean formula can be satisfied. Many combinatorial problems can be represented in the form of Boolean formulae and, thus, SAT is often used for solving these problems. SAT programs usually take the form of Conjunctive Normal Form (CNF) formulae. CNF is a simple language comprised of literals, that is, Boolean variables (x) or their negation ($\neg x$), joined together in disjunctive clauses (at least one literal in each clause must evaluate to true).

SAT programs are solved by recursively assigning variables and propagating the result. When propagation reaches failure (by the presence of an empty clause) a naive SAT solver will backtrack by reverting the most recent decision. This is how the traditional Davis Putnam Logemann Loveland (DPLL) algorithm works (Davis et al., 1962). An extension to this, referred to as conflict driven clause learning (CDCL), attempts to make more intelligent decisions when backtracking. The approach operates similarly to the DPLL algorithm until a decision causes failure. When this happens, it constructs an implication graph, the analysis of which can allow the solver to construct a clause that succinctly describes the cause of failure. This clause is known as a nogood. Using the nogood, the solver can *backjump* to the decision that lead to the failure, avoiding the exploration of many unrelated decisions. This can yield considerable speed-ups (Silva et al., 1996).

The simplicity of the language and the power of nogood learning makes SAT a powerful solving technology, capable of dealing with millions of variables and clauses. However, despite its ability to solve large programs, the simplicity of the language also means that even simple problems might require large, complex program formulations, and require considerable modelling expertise. Reducing the number of clauses and variables has been found to translate directly into faster solving times. Therefore, reducing the size of these formulations is important.

There are many solvers available for solving SAT problems. They can be loosely categorised into two classes: those applying the CDCL approach, such as MiniSat (Eén et al., 2004) and Chaff (Moskewicz et al., 2001), and those utilising some form of local search, such as WalkSat (Selman et al., 1995).

Boolean satisfiability example. Implementing a SAT program for a problem involves representing all decisions in terms of Boolean variables. There are several ways to represent integer decisions with Boolean variables. One way is to introduce a Boolean variable for each possible value an integer variable can take (e.g., $b_{X=5}$ represents the assignment of 5 to the integer decision X), as with binary indicator variables in MIP. This is the approach taken in the case of both the direct and support encodings (Gent, 2002; Walsh, 2000).

In the direct encoding, constraints are represented by enumerating the assignments that the constraint rejects. For example, if a constraint states that the assignments $X = i$

$$\begin{array}{ll}
\text{given: } m & \text{integer} \\
& (2.11) \\
\text{subject to: } \bigvee_{k=0}^{m^2} \text{mark}_{ik} & \forall_{i=1}^m \\
& (2.12) \\
\neg \text{mark}_{ik_1} \vee \neg \text{mark}_{ik_1} & \forall_{i=1}^m \forall_{k_1, k_2=0, k_1 < k_2}^{m^2} \\
& (2.13) \\
d_{ijk} \leftrightarrow \bigvee_{k_1=0}^{m^2} \bigvee_{k_2=0, k=k_2-k_1, k_1 \neq k_2}^{m^2} (\text{mark}_{ik_1} \wedge \text{mark}_{jk_2}) & \forall_{i,j=1, i < j}^m \forall_{k=0}^{m^2} \\
& (2.14) \\
\bigwedge_{i_2=1, j_2=1, i_2 < j_2, i_1 < i_2 \vee j_1 < j_2}^m (\neg d_{i_1 j_1 k} \vee \neg d_{i_2 j_2 k}) & \forall_{i_1=1, j_1=1, i_1 < j_1}^m \\
& (2.15) \\
d_{ijk} \in \{\text{true}, \text{false}\} & \forall_{i=1}^m \forall_{j=1}^m \forall_{k=0}^{m^2} \\
& (2.16) \\
\text{mark}_{ik} \in \{\text{true}, \text{false}\} & \forall_{i=1}^m \forall_{k=0}^{m^2} \\
& (2.17)
\end{array}$$

Figure 2.2: SAT formulation of the Golomb Ruler Problem.

and $Y = j$ are incompatible, the SAT clause is $\neg b_{X=i} \vee \neg b_{Y=j}$. In contrast, in the support encoding constraints are represented by enumerating the assignments that the constraint admits. Clauses are added that state for each pair of variables X and Y in a constraint, that either variable X is not assigned some value, or one of the supporting values for variable Y must be taken: $\neg b_{X=i} \vee (\bigvee_j b_{Y=j})$.

For example, encoding the *mark* variables from the Golomb Ruler problem in the direct or support encoding, requires the introduction of m^2 Boolean variables for each integer variable, with mark_{ik} representing mark i being in position k . Since each mark can be in only a single position, a clause is added representing an **at-least-one** constraint: $\forall_{i=1}^m \bigvee_{k=0}^{m^2} \text{mark}_{ik}$. In addition, a set of clauses representing an **at-most-one** constraint must be posted since a mark cannot be in multiple positions: $\forall_{k,l \in 0..m^2, k \neq l} (\neg \text{mark}_{ik} \vee \neg \text{mark}_{il})$

Another possible encoding for variables and constraints is that of the order encoding (Ansótegui et al., 2005), where Boolean variables are added that represent bounds on integer decisions instead of simple assignments. Let us again illustrate this by encoding the mark_2 variable from the Golomb Ruler problem above. Boolean variables representing an upper bound on the value that mark_2 can take are added to the program ($b_{\text{mark}_2 \leq 0}, b_{\text{mark}_2 \leq 1}, \dots, b_{\text{mark}_2 \leq m^2}$). Just as in the case of direct and support encoding of variables, these variables require a set of clauses that encode the relationship between the different variables: $\forall_{i \in 0..(m^2)-1} (\neg b_{\text{mark}_2 \leq i} \vee b_{\text{mark}_2 \leq i+1})$. This is a linear number of clauses, in comparison to the quadratic number of clauses required by the **at-most-one** constraints of the direct and support encodings. With this encoding, representing a constraint such as $\text{mark}_2 < 4$ is as simple as posting the unit clause $b_{\text{mark}_2 \leq 3}$ and discarding

the variables representing higher valued assignments. This encoding can make certain kinds of constraints easier to represent.

Figure 2.2 presents a high-level SAT formulation of the Golomb Ruler problem. In this formulation, *mark* represents the marks using the direct encoding of the mark integer decisions. Equation 2.12 encodes the **at-least-one** constraints on the *mark* variables as at least one value must be selected for each integer decision. Equation 2.13 encodes the **at-most-one** constraints which makes sure that no attempt is made to select more than one value for any integer decision. Just as in the case of the MIP formulation above, the distances between marks are represented by variables named *d*. In this case, *d* is a three-dimensional matrix where $d_{ijk} = \text{true}$ states that the distance between the marks mark_i and mark_j is *k*. This is encoded by Equation 2.14. This constraint would have to be translated a little further to be valid in a CNF program. Equation 2.15 then represents the **alldifferent** constraint on the distances. Note that there is no **minimize** objective here, as optimisation in SAT is typically controlled from outside of the CNF program. The solver is used to find a valid solution first, then a bound is set on the objective and a new formulation is generated with this bound added. For $m = 4$, this formulation results in a SAT program containing roughly one thousand variables and four thousand clauses.

2.3.4 Constraint Programming

Constraint programming (CP) is another powerful approach for modelling and solving combinatorial problems. Modern CP comes as the result of many years of research, from early constraint based systems (Sutherland, 1964), to constraint logic programming in Prolog (Emden et al., 1976), to constraint based reasoning, which introduced the general concept of a Constraint Satisfaction Problem (Güsgen et al., 1992) and, finally, to the modern concept of Constraint Programming (Rossi et al., 2006). A constraint program is comprised of a set of variables, domains, constraints and an objective function to be optimised (if any), much like the other approaches. The main difference resides in the wide variety of types of variables and constraints supported. Typically, CP solvers provide many high-level constraints over variable types such as Booleans, integers, floats, and sets, ranging from simple binary relations to complex global constraints describing entire sub-problems involving many variables. Constraint solving systems, such as Gecode (Gecode Team, 2006) support user defined variable and constraint types. This generality allows CP solvers to solve problems described using a high-level, compact representation, more suitable for non-expert users.

Many CP solvers exist, including Mistral (Hebrard, 2016), Minion (Gent et al., 2006a), Opturion CPX (Opturion, 2016), Chuffed (Chu, 2011), Choco (Prud'homme et al., 2016), and IBM CP Optimizer (IBM, 2016). They solve combinatorial problem by interspersing *constraint propagation* and *search* during solving.

Constraint Propagation

One of CP's greatest strengths is the idea of constraint propagation. A propagator for a constraint is a specialised algorithm that removes values from the domains of variables

that will never satisfy a constraint. For example, consider the variables $0 \leq x, y \leq 10$ and a constraint $x < y$. The *less-than* propagator will remove 10 from the domain of x , and 0 from the domain of y since these values cannot take part in a solution. Propagators provide a much more general interface than that of the constraints in MIP or SAT, and can be defined over many different kinds of constraints and variables. This is what allows CP to support a much more expressive language. Constraint solvers will typically execute propagators for each constraint repeatedly until they no longer change the domains of any variables, that is, until fixed-point.

Global constraints. Propagators can operate on large sets of variables, and implement more complex constraints than simple inequality or logical relationships. In particular, many structures that occur as subproblems of different problems often have dedicated algorithms implemented for them; these are called *global constraints*. The best known global constraint is the *alldifferent* constraint, which ensures that the variables in a set take different values in a solution. As such, the global constraint concisely expresses high-level information regarding the structure of the problem, and can be seen as a shorthand for the semantically equivalent relation that can be expressed as the conjunction of *not-equal* constraints.

Using global constraints grants three main advantages. The first is that they allow problems to be described in a more clear and concise way. For example, one could use a large set of simple constraints to ensure that a variable x should be assigned the index of the variable in array a that has the smallest value. Alternatively, one could simply use the *minimum-arg* constraint, which describes this relationship more succinctly.

The second advantage is that solvers can implement dedicated propagators for these global constraints allowing for stronger propagation, that is, propagation that would not occur given the simpler decomposition. For example, a set of *not-equal* constraints being propagated in isolation will not provide the same level of propagation possible with a dedicated *alldifferent* propagator. The case of Hall-sets show why this is the case. Assume a problem has three variables x, y and z with respective domains $\{1, 2, 3\}, \{1, 2\}, \{1, 2\}$ and a clique of *not-equal* constraints between each pair of variables (i.e., $x \neq y, x \neq z, y \neq z$). Local propagation of each of these constraints in isolation has no impact on the domains, since we cannot safely remove any values until one of the variables has been assigned. Conversely, if the clique is represented by a single global constraint (*alldifferent*), the propagation algorithm can assign x to 3 since the values 1 and 2 must be taken by y and z , leaving 3 as the only valid assignment to x . Propagation algorithms for *alldifferent* have been introduced by Leconte (1996), López-Ortiz et al. (2003), Puget (1998), and Régin (1994). An empirical survey of some techniques for implementing the *alldifferent* propagator is presented in Gent et al. (2008).

The third advantage of using global constraints is that they allow compilers for high-level modelling languages to make better decisions during compilation, when the structure represented by global constraints is made explicit in the model, as will be shown in Section 2.4.1.

Common global constraints. The Global Constraint Catalog maintains a large catalogue of definitions and implementations of global constraints and their availability in different constraint programming solvers (Beldiceanu et al., 2007). The following briefly discusses a selection of global constraints that appear frequently in constraint programs and are used later in this thesis.

alldifferent The `alldifferent` constraint restricts a set of variables to take distinct values.

table The `table` constraint allows constraints to be described extensionally, that is, as a list of tuples that represent valid (supporting) assignments to variables. This constraint requires the enumeration of the solutions of a constraint, which can be inefficient but is sometimes worth the extra cost (Dekker, 2016).

cumulative The `cumulative` constraint is a common constraint in scheduling problems. It maintains that a schedule for a set of tasks, with start times, durations, and resource demands, can be found that does not exceed some resource bound in any time step.

lex-less The `lex-less`, `lex-greater`, and the associated `lex-less-eq` and `lex-greater-eq` enforce a lexicographic ordering between two arrays of variables. These are often used as symmetry breaking constraints.

element The `element` constraint allows the indexing of an array using a variable as index. For example, $x = A[v]$ where x and v are variables and A is an array.

global-cardinality The `global-cardinality` constraint maintains that certain values must occur a certain number of times in an array.

bin-packing The `bin-packing` constraint packs a set of objects of different weights (or sizes) into a set of bins, ensuring that the capacities of the bins are not exceeded.

sliding-sum The `sliding-sum` constraint states that within a sequence, the sum of any consecutive sequence of elements lies between a certain lower and upper bound.

knapsack The `knapsack` constraint takes a set of item weights and profits, and maintains a connection between these and variables representing the number of items that are selected, the sum of the weights of selected items, and the sum of the profit from the selected items.

Search

CP solvers typically use a backtracking depth-first search to solve constraint satisfaction programs (that is, problems without an objective function). For combinatorial optimization problems, they add to this some form of branch and bound similar to that of MIP solvers (see below).

A backtracking depth-first search typically proceeds as follows: First, propagation is called. This initial call to propagation is known as *root-node propagation*, as it is performed at the root of the search tree. The propagation performed at the root node is important

because the inferences found at the root (before any decisions have been made) are valid for the rest of the search, no matter how it proceeds. If propagation wipes out (empties) the domain of a variable at the root-node, the constraint program is trivially unsatisfiable, meaning there is no valid assignment to the variables that will satisfy all constraints simultaneously. Otherwise, after root-node propagation, search will begin to recursively select variables and assign them values from their domains. Variables and values are selected using one of many different available heuristics. Most solvers have several default variable selection heuristics such as *lexical selection*, which selects variables in the order they appear in the program, and *first-fail selection*, which selects the variable with the smallest domain in an attempt to prune the search tree as early as possible. Many other heuristics can be employed and it is often the case that the user may implement their own problem-specific heuristics for a solver to use. In addition to the selection of a variable, a value from that variable's domain must also be selected for it to be assigned to. This is achieved using what is called a value selection heuristic.

Once all variables are assigned (all domains have only one value left) the solver has found a satisfying solution to the problem. If at any time, however, a variable has an empty domain, it cannot take a value and, thus, the subproblem is unsatisfiable. In this case, solvers will typically backtrack to the last variable/value selected and select a different one. If the algorithm backtracks out of all branches from the root node without ever finding a satisfiable solution, the program can be declared unsatisfiable.

For optimization problems, the depth-first search approach mentioned above is often extended to perform *branch-and-bound* in such a way as to optimise the objective function. Typically, an objective variable is added to the program together with a constraint ensuring this variable takes the value of the objective function. If a solution is found during search, the value of the objective variable is retrieved and stored as the current best bound. In addition, a constraint is added to the program stating that the objective variable must take a value better than this bound. Search continues as normal and the bound is updated whenever a better solution is found. When the solver finishes the exploration or pruning of the remaining search space, it has *proven optimality* or, if no solution has been found, unsatisfiability.

Constraint Programming Example

Figure 2.3 presents a possible constraint model for the Golomb ruler problem. The model has a single parameter m denoting the number of marks required on the ruler. Each mark $mark_i$ must take a value somewhere within the range 0 to m^2 . A constraint is posted stating that each consecutive pair must be ordered (Constraint 2.21). An *alldifferent* global constraint ensures that the distance between every pair of marks (computed by 2.22) is distinct (Constraint 2.23). Also, a constraint fixing the first mark ($mark_1$) to zero is added, as we did for the MIP program (Constraint 2.20). In contrast to the MIP and SAT formulations, for $m = 4$, this formulation results in a program containing only 16 variables and 6 constraints.

given:	m	integer	(2.18)
minimize:	$mark_m$		(2.19)
subject to:	$mark_1 = 0$		(2.20)
	$mark_i < mark_{i+1}$	$\forall_{i=1}^{m-1}$	(2.21)
	$mark_j - mark_i = d_{ij}$	$\forall_{i=1}^m \forall_{j=i+1}^m$	(2.22)
	$\text{alldifferent}(\cup_{i=1}^m \cup_{j=i+1}^m d_{ij})$		(2.23)
	$0 \leq mark_i \leq m^2$	$\forall_{i=1}^m$	(2.24)
	$0 \leq d_{ij} \leq m^2$	$\forall_{i=1}^m \forall_{j=i+1}^m$	(2.25)

Figure 2.3: Constraint Model of the Golomb Ruler problem.

It is clear from this example, that the constraint programming approach leads to much more natural, concise descriptions of problems. This is why the language of constraint programming is nowadays used as the basis of major high-level modelling languages, with problems being described as constraint programs and then compiled into the more low-level MIP or SAT programs if required.

2.4 High-Level Modelling

Traditionally, problems were modelled by writing code that would construct input for a solver. Later with systems based around constraint logic programming (Emden et al., 1976), a declarative language was used to describe instances of a problems which are then passed to one of several back-end solvers. Later, stand-alone modelling languages became available. One such language is AMPL (Fourer et al., 1989), which provided a language for describing MIP models which could be combined with data and then be compiled into standard input formats supported by MIP solvers. OPL (Van Hentenryck et al., 1999) is another high-level modelling language that allowed problems to be modelled and solved by both MIP and CP solvers. OPL had a fixed set of types and constructs that could be used for describing problems but was not easily extended. Zinc (Garcia de la Banda et al., 2006) is a very high-level language that supported arbitrarily nested user-defined types, predicates, and functions, making it a very powerful language for describing problems. ESSENCE (Frisch et al., 2007) is another very high-level modelling language that supports arbitrarily nested types with a focus on describing CP models in terms of first-order relations. MiniZinc (Nethercote et al., 2007), is a high-level language that was later introduced a simplified and more practical modelling language than Zinc. The most notable difference between MiniZinc and Zinc is the removal of the arbitrarily nested user-defined types. MiniZinc allows combinatorial problems to be modelled and solved using a wide selection of solvers. The language used to describe problems is independent of the solver used and, thus, problems are typically modelled as CP models since CP provides the most flexible language for describing problems. The challenge for a compiler then, is to compile these solver-independent models from high-level constraint models to efficient

```

1  % golomb.mzn
2  int: m;
3
4  array[1..m] of var 0..m*m: mark;
5  constraint mark[1] = 0;
6
7  constraint forall ( i in 1..m-1 )
8      (mark[i] < mark[i+1]);
9
10 constraint alldifferent(
11     [mark[j]-mark[i] | i in 1..m,
12                        j in i+1..m]);
13
14 solve minimize mark[m];

```

<pre> % 4.dzn m = 4; </pre>	<pre> % 10.dzn m = 10; </pre>
-----------------------------	-------------------------------

Figure 2.4: MiniZinc model and two data files for the Golomb Ruler problem.

programs for whatever target solver the user selects. To better formalise this challenge, this section first introduces MiniZinc using the Golomb ruler example and then presents a reduced MiniZinc grammar, adapted in part from Stuckey et al. (2013) and Feydy et al. (2011). This reduced grammar will be used later to illustrate the translation of high-level MiniZinc into a low-level representation, called FlatZinc, consisting of a flat set of variables and constraints, suitable for a particular target solver.

2.4.1 MiniZinc by Example

Figure 2.4 shows a MiniZinc model for the Golomb ruler problem presented in Figure 2.3. Line 2 in the model declares the integer parameter `m` by leaving it unassigned. Without a value for all parameters, a MiniZinc model cannot be compiled. Line 4 in the model declares an array `mark` of variables (indicated by the **var** keyword) which can take integer values in the range 0 to m^2 inclusive. This is followed by the first constraint which simply forces the first mark to be at position 0. Lines 7-8 constrain consecutive marks to be strictly ordered. Lines 10-12 construct a new array of variables, using an array comprehension (`[... | ...]`), to hold the distances between each pair of marks, and then constrains this array of variables to take distinct values using the `alldifferent` global constraint. Finally, the objective is given in Line 14 as the minimisation of the position of the final mark `mark[m]`. Two example data files can be seen under the model providing different assignments to parameter `m`.

Note that while the model mostly follows the one presented in Figure 2.3, there are some minor differences. Primarily, the set of variables d_{ij} does not need to be explicitly stated in the MiniZinc model, as they are introduced by the array comprehension which is passed to the `alldifferent`.

2.4.2 Grammar of a Simplified MiniZinc

The MiniZinc modelling language supports several variable types (Boolean, Integer, Float, Set), and has a large library of constraints to choose from when modelling. As seen in the example above, a constraint model in MiniZinc contains many different kinds of items including declarations of parameters and decision variables, constraints expressed in terms of loop constructs such as `forall` and `exists` quantifiers, array/set comprehensions, and a solve item which can specify an objective function to be minimised/maximised, or specify the problem as a satisfaction one. More complex models can contain nested expressions and other kinds of items such as function and predicate definitions and calls.

MiniZinc declarations and items. While there are many different types of items in the full MiniZinc specification, here we focus on those used to define predicates, state the constraints of a problem, and state the type of problem (satisfaction or optimisation, with an objective function). This is because these items are the only ones needed to illustrate how compilation of MiniZinc works. The following set of production rules define the declarations and items allowed in a simplified MiniZinc model.

```

model  →  decls items
items  →  ε | item items
item   →  predicate pred ( params ) = cons ; | predicate pred ( params );
        →  constraint cons; | solve sense;
params →  ε | bdecl | bdecl, params | idecl | idecl, params
sense  →  satisfy | minimize obj | maximize obj
obj    →  ivar
decls  →  ε | idecl; decls | idecl = term ; decls
        →  bdecl ; decls | bdecl = cons ; decls
        →  array[int] of idecl; decls | array[int] of idecl = iarray ; decls
        →  array[int] of bdecl; decls | array[int] of bdecl = barray ; decls
idecl  →  int : ivar | var int : ivar | term..term : ivar | var term..term : ivar
bdecl  →  bool : bvar | var bool : bvar

```

These rules state that a `model` has two components: `decls`, followed by `items`. The latter can either be ϵ (empty) or an `item` followed by further `items`. An `item` is then described as being either a predicate declaration, a constraint or a solve item. The production rules for a `cons` item will be described in the next section. Two forms of predicate declaration are presented. Common to both is `params` which, combined with `pred` (the name of the predicate), represent the signature of the predicate. The first form of predicate declaration has a `cons` as its right-hand-side. This `cons` provides the definition (i.e., decomposition) of the predicate. The second form does not have this right-hand-side, indicating that the predicate is a builtin and should not be decomposed. In the simplified MiniZinc, only integer and Boolean variables and parameters are represented. The full MiniZinc language also supports set and float variables. The last two production rules for an `item` introduce constraints (using the `cons` seen before for predicates) and the `solve` item, which can be declared with three “senses”: `satisfy` marks the problem as a satisfaction one where any

solution will do, while `maximize` or `minimize` mark it as an optimisation problem for an objective function with value given by an integer variable.

We next see the production rule for `decls`, which introduces the variable and parameter declarations for the model. Note that integer declarations (`idecl`) can be bound to a `term` (an expression returning a scalar) and Boolean declarations can be bound to a `cons` (a Boolean expression). Arrays of Boolean and integer variables and parameters can also be declared. These can also be bound to literal Boolean and integer arrays (`barray` and `iarray`). Finally, we have the two production rules for parameter and variable integer and Boolean declarations. Integer declarations can also take a set, denoted by `term..term`, with integer expressions that represent the bounds of the variable.

MiniZinc expressions. MiniZinc expressions are comprised of `cons` and `term` items. The former represent Boolean expressions, that is, constraints in the root context and reified constraints otherwise. The latter represent integer expressions, that cannot occur in the root context; only as arguments to `cons`. The following presents the rules for `cons` and `term` expressions, along with those for Boolean and integer arrays (and comprehensions).

<code>cons</code>	\rightarrow	<code>true</code> <code>false</code> <code>bvar</code> <code>term relop term</code>
	\rightarrow	<code>not cons</code> <code>cons relop cons</code>
	\rightarrow	<code>barray[term]</code> <code>forall barg</code> <code>pred(args)</code>
	\rightarrow	<code>if cons then cons else cons endif</code> <code>let decls in cons</code>
<code>relop</code>	\rightarrow	<code>=</code> <code>!=</code> <code><</code> <code><=</code> <code>-></code> <code><-></code>
<code>args</code>	\rightarrow	<code>ε</code> <code>iarg, args</code> <code>barg, args</code>
<code>barg</code>	\rightarrow	<code>bvar</code> <code>barray</code>
<code>iarg</code>	\rightarrow	<code>ivar</code> <code>iarray</code>
<code>term</code>	\rightarrow	<code>int</code> <code>ivar</code> <code>term arithop term</code> <code>iarray[term]</code>
	\rightarrow	<code>if cons then term else term endif</code>
<code>arithop</code>	\rightarrow	<code>+</code> <code>-</code> <code>/</code> <code>*</code> <code>mod</code>
<code>barray</code>	\rightarrow	<code>[cons, ..., cons]</code> <code>[cons_v v in term..term]</code>
<code>iarray</code>	\rightarrow	<code>[term, ..., term]</code> <code>[term_v v in term..term]</code>

The first rule for `cons` shows that it can be a Boolean value (`true` or `false`), a Boolean variable or a relational operator (such as `<`, `>`, `=`, and `!=`) between two `terms`. In addition, a `cons` can also take the form of a negation (`not cons`), or a logical operator (such as `/\`, `\`, `->`, and `<->`) between two `cons`. Next we have the indexing of a Boolean array by a `term`. This will return the element in the array, selected by the `term`. After this we see that `cons` can be a `forall` which takes an array of Booleans as its argument and is true if, and only if all Booleans are true. Another kind of `cons` is that of a predicate call. Here instead of the `params` from a predicate declaration item, we have `args` which the production rule indicates can be empty, a sequence of integer variables, Boolean variables, integer arrays or Boolean arrays. The next production rule for `cons` is that of an if statement. Since it is a `cons` item, the `then` and `else` branches must also be `cons`. The last production rule for `cons` is a `let` expression (`let decls in cons`), which introduces the variables declared in `decls` and uses these declarations in `cons`. Next we have the production rules for `term` expressions. Terms are expressions that resolve to non-Boolean values (only integers in the simplified MiniZinc presented here). A `term` can be any integer parameter or variable, an arithmetic

operation between two terms (+, -, /, and *) or an integer array access (`iarray[term]`). Finally, the `if` expression can be used with `terms` as the `then` and `else` branches.

2.5 Compilation

Recall Figure 1.2 which showed the modelling, compilation, and solving process common to modern modelling languages for combinatorial problems. The user starts the process by formalising their problem as a model, that is, a parametric specification of the problem. Following this, the compiler combines the model with a given set of data to form an instance, that is, a non-parametric model. Conceptually the instance is a useful stage to reason about, as all the instance specific information is available but the compiler is yet to make any decisions about how the instance will be further compiled to a program. The compiler then translates the instance into a program that can be understood and solved by the target solver.

2.5.1 Compiling MiniZinc Instances to Programs: an Overview

MiniZinc follows this same process, with high-level (parametric) MiniZinc models being translated into concrete solver-level programs, comprised of a flat list of variable declarations and *builtin* constraints, that is, constraints supported natively by the solver. This translation, called *flattening*, unrolls all loops and reduces expressions to sets of individual, atomic, flat constraints that are part of the FlatZinc language. In this thesis the terms “flattening” and “compiling” are used interchangeably. The FlatZinc language is a subset of MiniZinc comprising only variable and constant declarations, along with simple constraints. In contrast to MiniZinc, FlatZinc has no loops or complex expressions, and only supports optimisation of a single variable (an objective function must be encoded using constraints to constrain the objective variable). All constraints in a FlatZinc program are represented by calls to builtin predicates. These take the form: `predicate_name(arguments)`.

Solver writers need only provide support for reading FlatZinc in order to support the solving of combinatorial problems defined using MiniZinc. This can be achieved by either implementing a stand-alone parser, which translates FlatZinc programs into the correct internal representation for solving or, alternatively, by implementing a solver interface to the `libminizinc`¹ (Tack, 2011) library, which already provides a MiniZinc (and, thus, FlatZinc) parser. In addition, it provides type checking and flattening (compilation) together with several existing solver interfaces. Thus, implementing a solver interface for the `libminizinc` library is a simple way to add MiniZinc support to a solver.

Each predicate in the constraint library can be declared in two different ways: without a definition (marking it as a builtin), or with a definition that defines it in terms of simpler constraints (marking it as a *decomposition*). Therefore, for each target solver, the MiniZinc system must contain a specialised library that declares all of its builtins and

¹<http://www.minizinc.org>

```

1 include "all_different_int.mzn";
2 include "all_different_set.mzn";
3
4 % Constrain the array of integers x to be all different.
5 predicate all_different(array[int] of var int: x) =
6   all_different_int(x);
7
8 % Constrain the array of sets of integers x to be all different.
9 predicate all_different(array[int] of var set of int: x) =
10  all_different_set(x);

```

Listing 2.1: all_different.mzn.

```

1 % Constrains the array of objects 'x' to be all different.
2 predicate all_different_int(array[int] of var int: x) =
3   forall(i,j in index_set(x) where i < j) ( x[i] != x[j] );

```

Listing 2.2: all_different_int.mzn.

provides solver specific decompositions for constraints that are not natively supported, falling back on the decompositions from the standard library if necessary.

Different solvers require problems to be described in different ways. However, they can be relatively uniform within a particular solver class. For example, all MIP solvers support only integer and continuous variables with contiguous domains, and linear constraints. Thus, the MiniZinc distribution comes with a library for MIP solvers called `linear`, which provides linear decompositions for many constraints such as global and non-linear constraints. Similarly, all SAT solvers support only Boolean variables and simple clauses. A slight modification of the MIP library developed in Abío et al. (2014) is used as the basis of a SAT translation for MiniZinc. These two libraries allow high-level constraint programming style models in MiniZinc to be solved by any MIP or SAT solver.

When it comes to CP solvers however, there is more variation. For example, some CP solvers support continuous variables, while others support set variables, and no two solvers support the exactly the same set of global constraints. As such, each CP solver must provide a library declaring which global constraints are supported and which must be decomposed, or possibly provide decompositions that happen to better suit the solver. As an example, consider a model that uses the constraint `set_eq(s1,s2)`, which constrains the two set variables (`s1` and `s2`) to be equal. A solver that does not natively support set variables could provide a library definition for this constraint that introduces Boolean variables to represent the two set variables, and a set of constraints on these that describe the `set_eq` constraint. If the connection between the set variables and their representative Boolean variables is functional, and they do not occur in any other constraints, they can be left out of the program. The set variables and constraints that link them to the Boolean variables in the program, would be stored in a separate file (`.ozn`) and used later for outputting solutions.

As another example, consider Listing 2.1 and Listing 2.2, which are extracts from the MiniZinc standard library of decompositions (`std`). In Listing 2.1 we see two definitions of a constraint `all_different` defined with two different signatures: the first for an array of integer variables and the second for an array of set variables. These two constraints get

```

1 var 1..10: a; var 1..10: b;
2 var 1..10: c; var 1..10: d;
3 constraint a * b + c = d;
4 constraint alldifferent([a, b, c, d]);
5 solve minimize d;

```

Listing 2.3: Sample Instance to be Flattened.

```

1 var 1..9: a; var 1..9: b;
2 var 1..9: c; var 2..10: d;
3 var 1..9: t1;
4 constraint int_times(a, b, t1);
5 constraint int_plus(t1, c, d);
6 constraint all_different_int([a, b, c, d]);
7 solve minimize d;

```

Listing 2.4: Flattened Program.

rewritten into `all_different_int` and `all_different_set` respectively. Listing 2.2 shows the standard decomposition for `alldifferent` over integer variables, which is to represent it as a clique of `not-equals` constraints.

Flattening in MiniZinc is implemented as a recursive call-by-value (inside-out) translation. Intuitively, this works as follows. If a predicate or function call is nested as an argument of another, it is decomposed into a separate constraint, and the argument is replaced with an auxiliary variable holding the result of the call. For Boolean predicates that are not in a top-level conjunction, a reified version of the expression will be constructed and bound to an auxiliary Boolean variable. Reified constraints are important for allowing complex problems to be modelled. However, they are often less efficient than their top-level counterparts. For example, many reified versions of global constraints require a specialised propagator and, thus, many solvers only implement a few reified versions of global constraints. As a result, the MiniZinc compiler is forced to instead post a decomposition, which may be costly in some circumstances.

After flattening the arguments of a call, the compiler looks for a predicate or function declaration with the same type signature. If it is defined as a builtin, the compiler inserts it into the FlatZinc as is. Otherwise, a copy of its body is instantiated with the actual arguments of the call and then flattened. Further optimisations performed by the MiniZinc compiler are explored in Section 2.5.3.

Listing 2.3 and Listing 2.4 demonstrate the flattening of a simple MiniZinc instance to FlatZinc. After adding the variables `a`, `b`, `c`, and `d`, the compiler reaches the expression $a * b + c = d$. This is flattened by first flattening the subexpression $a * b$. To do so, a new variable is introduced, t_1 which will hold the result of the expression. This expression is represented as `int_times(a, b, t1)` in the program, which constrains the introduced variable to be equal to the product of `a` and `b`. Next, with $a * b$ substituted by t_1 , the remaining expression $t_1 + c = d$ must be flattened. The flat constraint that represents this expression is `int_plus(t1, c, d)`. Finally, the `alldifferent` constraint is flattened to an `all_different_int` constraint.

Algorithm 2.1 The `mzn2fzn` driver.

```

procedure mzn2fzn(modelFile, dataFile, libraryFile)
  library  $\leftarrow$  parse(libraryFile)
  model  $\leftarrow$  parse(modelFile)
  data  $\leftarrow$  parse(dataFile)
  return flatten(library  $\cup$  model  $\cup$  data)

```

2.5.2 Compiling MiniZinc Instances to Programs: a Procedure

Now that the reduced representation of the MiniZinc grammar has been introduced and an overview of the flattening process has been provided, it is time to explain the compilation procedure in more detail. The components of the procedure will be extended in later chapters, as needed, to illustrate how certain extensions can be implemented.

The `mzn2fzn` driver. The `mzn2fzn` driver compiles a MiniZinc model and its data, together with a selected target solver’s library, into a FlatZinc program suitable for solving with the target solver. Pseudocode for this procedure is presented in Algorithm 2.1.

The algorithm takes a path to a required MiniZinc model file (*modelFile*), its MiniZinc data file (*dataFile*) and a selected library (*libraryFile*). Each is parsed into a set of items and declarations (*library*, *model* and *data*, respectively) and merged into a single set, which can be seen as a complete instance and is then passed to the `flatten` procedure. Note that this is possible because library files are essentially MiniZinc models, and data files follow a subset of the MiniZinc model grammar (the subset required to provide values to parameters). In practice, users will directly or indirectly (through the MiniZinc IDE for example) call the `mzn2fzn` program. For example, to compile the Golomb ruler model for $m = 4$ with a MIP target solver, the command `mzn2fzn -G linear golomb.mzn 4.dzn` would be executed. This will produce a FlatZinc program, `golomb.fzn`, and an output file, `golomb.ozn`, which is used for translating solutions from the solver into the output format requested by the user (MiniZinc’s output functionality is out of scope and will not be explored here).

The `flatten` procedure. As expressed in Section 2.5, an instance must be flattened before it can be solved. A procedure for flattening an instance is presented in Algorithm 2.2. All procedures called from `flatten` have access to the global compiler state. The main components of the state are the set S , which represents the FlatZinc program, the stack *CallStack*, which stores the sequence of expressions the compiler is currently processing and, finally, the map *PredicateMap* which stores a mapping of predicate names to their definitions (decompositions) in either the model or the target library. A predicate declaration, lacking a definition, signifies a builtin, and is stored in the map as \perp . Note that the while the compiler presented here assumes that predicate declarations will occur before any calls to those predicates, the real MiniZinc does not have this restriction.

After initialising the global state, `flatten` then steps through each declaration or item in the *instance* and applies rules based on their type. Note that the real MiniZinc compiler

Algorithm 2.2 Flattening an instance.**Compiler State:** S : set ▷ FlatZinc program $CallStack$: stack ▷ Trace of expressions currently being processed $PredicateMap$: map: $name \rightarrow item$ ▷ Mapping of predicate names to definitions**procedure** $flatten(instance)$ $S \leftarrow \emptyset$ $CallStack \leftarrow \emptyset$ $PredicateMap \leftarrow \emptyset$ **for each** ($elem \in instance$) **do** **switch** $elem$ **case predicate** $p(t_1, \dots, t_2) = c_0$: $PredicateMap[p(t_1, \dots, t_2)] \leftarrow c_0$ **case predicate** $p(t_1, \dots, t_2); :$ $PredicateMap[p(t_1, \dots, t_2)] \leftarrow \perp$ **case idecl** | **bdecl**: $flatd(elem)$ **case constraint** c : $flatc(true, c)$ **case solve satisfy**: $S \leftarrow solve\ satisfy$ **case solve** $s\ t$: $flatt(newVar(v), t)$; $S \leftarrow solve\ s\ v$ **return** S

does not iterate through the *instance* in order, since a better compilation can be achieved by postponing the compilation of some constraints.

If *elem* is a predicate definition with a body (c_0) that defines how a predicate can be decomposed using MiniZinc, its definition is added to *PredicateMap* without any further flattening, since it is not being called here. If *elem* is a predicate definition without a body, the special symbol \perp is added to the *PredicateMap* indicating that this is a builtin predicate. If *elem* is a variable declaration, the procedure $flatd(elem)$ is called to introduce a variable to the FlatZinc program. If *elem* is a constraint item, the procedure $flatc$ is called on this item's *cons*. $flatc$ takes two arguments b and c , where b is a Boolean control variable tied to the satisfiability of the constraint c , that is, if c is satisfied (unsatisfied) b will be true (false). The $flatten$ procedure fixes b to be *true* for all its constraints, as all top level (root context) constraints must be satisfied. Next, the case of a **solve** item that states that the problem is a satisfaction problem. In this case, the item is added directly to the program. The final case is that of a **solve** item, where the argument s provides the sense of the problem. The objective must be flattened by introducing an objective variable, and using $flatt$ to bind the expression to the variable. The **solve** item is then added to the program with the objective replaced by the this new variable. Finally, the flattened set of constraints and variable declarations of the program S are returned.

Flattening variable declarations. The $flatd$ procedure is presented in Algorithm 2.3. Depending on the type of variable declaration the procedure introduces a new variable by calling $newVar$ (Algorithm 2.4) and restricts the domains as necessary. Note that v in Algorithm 2.3 does not correspond to an actual variable, but to an identifier that will be used in the creation of the actual variable, v' , that will appear in the flattened program. If required, the compiler will in turn process terms by calling $flatt$ and *cons* by calling $flatc$. These procedures are discussed below. When constraints are ready to be added to the set

Algorithm 2.3 Flattening a variable declaration.

```

procedure flatd(decl)
  CallStack.push(decl)
  switch decl
  case array[int] of var l..u : v:  $\forall_{i=1}^{|v|}$  newVar(vi); c  $\leftarrow$  l  $\leq$  vi  $\cup$  vi  $\leq$  u
  case var int: v = t: c  $\leftarrow$  flatt(newVar(v'), t)
  case var l..u : v: newVar(v'); c  $\leftarrow$  l  $\leq$  v'  $\cup$  v'  $\leq$  u
  case var l..u : v = t: c  $\leftarrow$  flatt(newVar(v'), t)  $\cup$  l  $\leq$  v'  $\cup$  v'  $\leq$  u
  case var bool: b = c0: newVar(b'); flatc(b', c0)
  postConstraints({c})
  CallStack.pop()

```

of flattened FlatZinc constraints, the procedure **postConstraints** (Algorithm 2.5) is called and the *CallStack* popped. Note that, for brevity, this simplified MiniZinc only deals with variable bounds (thus, disregarding non-contiguous sets as domains).

Algorithm 2.4 Adding new variables.

```

procedure newVar(v)
   $\bar{v} \leftarrow$  new v
  return  $\bar{v}$ 

```

Algorithm 2.5 Adding constraints.

```

procedure postConstraints(C)
  for each (c  $\in$  C) do
    S  $\leftarrow$  S  $\cup$  {c}

```

The procedures **newVar** and **postConstraints** are quite simple, but are reproduced here as they will be expanded upon in later chapters. In particular, **newVar** will be extended in Chapter 4 to record the reasons for the introduction of different variables in support of a multi-pass compilation. The **postConstraints** procedure will be extended in a similar way in Chapter 6 to support the annotation of FlatZinc constraints with their reason for being introduced.

Flattening arrays. Algorithm 2.6 presents two helper procedures that will be used by the other flattening procedures. The first, **flate**, is used whenever the types of expression to be flattened are mixed (might be **term** or **cons**), e.g., in arguments to a predicate call. The procedure takes two arguments: *v* and *e*. If *e* is a **term**, the procedure will call **flatt** to bind *v* to the result of flattening *e*. Alternatively, if *e* is a **cons**, **flatc** will be called and *v* will be used as the control variable for the flattened expression *e*.

Next, a procedure for flattening array literals and comprehensions is presented called **flata**. This procedure takes two arguments: a control variable *b*, and an array literal or a comprehension *t*. If *t* is an array literal, the algorithm first checks the value of *b* and, if it is known to be *true*, it flattens each element as a **cons** expression in the root-context. If the value of *b* is unknown or undefined (argument passed as: $_$) each element *t_i* of the array is flattened using **flate**, with the resulting introduced variable being stored in *t'_i*. If given an array comprehension with expression *e*, variable *v* and range *t₁* .. *t₂*, the procedure expands the comprehension, flattening by assigning values from the range to *v* in *e* and flattening this instantiated expression. To do this, the lower and upper bounds must be computed. The **eval** function takes a fixed expression and returns its value. For each index value *i* in the calculated range, the assignment *v* = *i* is pushed onto the *CallStack*.

Algorithm 2.6 Helper procedures for flattening expressions and arrays.

```

procedure flate( $v, e$ ) ▷ Helper for flattening cons or term
  if  $e$  is a term then return flatt( $v, e$ ) else return flatc( $v, e$ )

procedure flata( $b, t$ )
  switch  $t$ 
  case  $[t_1, \dots, t_n]$ :
    for each  $i \in 1 \dots n$  do
      if  $b \equiv \text{true}$  then flatc( $\text{true}, t_i$ ) else flate(newVar( $t'_i$ ),  $t_i$ )
  case  $[e \mid v \text{ in } t_1 \dots t_2]$ :
    for each  $i \in \text{eval}(t_1) \dots \text{eval}(t_2)$  do
      CallStack.push(assignment  $v = i$ ) ▷ Assignment should be recorded
      if  $b \equiv \text{true}$  then flatc( $\text{true}, e[v/i]$ ) else flate(newVar( $t'_i$ ),  $e[v/i]$ )
      CallStack.pop()
  if  $b \equiv \text{true}$  then  $b$  else return  $[v'_1, \dots, v'_n]$ 

```

This allows the compiler to produce more useful feedback when printing errors, as it can show which specific iteration of the comprehension caused a failure. Next, the assignment has to be substituted into the expression e by replacing v with i ($e[v/i]$), which is then flattened as before. The assignment is popped from *CallStack* and the loop continues to work through the other assignments to v . Finally, if b is *true*, it is returned. Alternatively, the result of flattening the elements of the array literal or expanding the comprehension is returned.

Flattening constraints. Algorithm 2.7 shows how constraints are recursively flattened by the MiniZinc compiler using *flatc*. The procedure takes two arguments, b and c , with b being a Boolean variable representing the result of the evaluation of constraint c . Recall that calls to *flatc* from the *flatten* procedure set b to *true*, as the constraints are in the root-context. The procedure starts by pushing the current constraint to be flattened onto the *CallStack*. Different cases are then applied depending on the type of constraint being flattened. The first two cases flatten constraints in the form of Boolean literals. The control variable b is posted directly to the constraint set C , either in a positive or negated form. The next case flattens a Boolean variable b' . In this case we unify the control variable b with b' so that any further use of either returns the same variable. Note that the full MiniZinc compiler binds identifiers (use of variables) to actual variables and, thus, unification points two identifiers at the same variable, and sets the domain of the variable to be the intersection of the two original variables. The next case flattens a relational constraint between two terms. This is flattened by first calling *flatt* to flatten the terms and introduce new variables i_1 and i_2 to represent them. Then, *flatc* is called on a predicate form of the relational operator ($r(i_1, i_2)$) which can be posted or further flattened, if needed.

The next case is that of a negated constraint (**not** c_1). For this case, a new variable b_1 is introduced as the control variable for constraint c_1 , and a predicate `bool_not(b_1, b)` is posted. After this we have a case for handling a conjunction of two constraints. If the control variable b is known to be *true*, the flattener can simply flatten c_1 and c_2 in the

Algorithm 2.7 Flattening a constraint.

```

procedure flatc( $b, c$ )
  CallStack.push(c)
  switch  $c$ 
  case true:  $C \leftarrow \{b\}$ 
  case false:  $C \leftarrow \{\neg b\}$ 
  case  $b'$  (bvar):  $C \leftarrow \text{unify}(b, b')$ 
  case  $t_1 \text{ } r \text{ } t_2$ :  $C \leftarrow \text{flatt}(\text{newVar}(i_1), t_1) \cup \text{flatt}(\text{newVar}(i_2), t_2)$  ; flatc( $b, r(i_1, i_2)$ )
  case not  $c_1$ : flatc( $\text{newVar}(b_1), c_1$ ) ; flatc(bool_not( $b_1, b$ ))
  case  $c_1 \wedge c_2$ : if  $b \equiv \text{true}$  then flatc(true,  $c_1$ ) ; flatc(true,  $c_2$ )
    else flatc( $\text{newVar}(b_1), c_1$ ) ; flatc( $\text{newVar}(b_2), c_2$ ) ; flatc( $b, \text{bool\_and}(b_1, b_2)$ )
  case  $c_1 \vee c_2$ : flatc( $\text{newVar}(b_1), c_1$ ) ; flatc( $\text{newVar}(b_2), c_2$ ) ; flatc( $b, \text{bool\_or}(b_1, b_2)$ )
  case  $p(t_1, \dots, t_n)$  (pred call):
    for each  $j \in 1 \dots n$  do ▷ Flatten arguments as  $v_1 \dots v_n$ 
      if  $t_j$  is an array then  $v_j \leftarrow \text{flata}(\_, t_j)$ 
      else flate( $\text{newVar}(v_j), t_j$ )
    if ( $\text{PredicateMap}[p] \neq \perp$ ) then ▷ Predicate has a body?
       $p(x_1, \dots, x_n) = c_0 \leftarrow \text{PredicateMap}[p]$ 
      flatc( $b, c_0[x_1/v_1, \dots, x_n/v_n]$ )
    else
      if  $b \equiv \text{true}$  then  $C \leftarrow \{p(v_1, \dots, v_n)\}$ 
      else flatc(true,  $p\_reif(v_1, \dots, v_n, b)$ )
  case if  $c_0$  then  $c_1$  else  $c_2$  endif: if eval( $c_0$ ) then flatc( $b, c_1$ ) else flatc( $b, c_2$ )
  case let {  $d$  } in  $c$ :
    let  $\bar{v}'$  be a renaming of variables  $\bar{v}$  defined in  $d$ 
    flatc( $b, \text{flatlet}(d[\bar{v}/\bar{v}'], c[\bar{v}/\bar{v}'], 0)$ )
  case forall  $ba$ :
     $[b_1, \dots, b_n] \leftarrow \text{flata}(b, ba)$ 
    if  $b \neq \text{true}$  then
       $C \leftarrow \{\text{array\_bool\_and}([b_1, \dots, b_n], b)\}$  ▷ Conjunction of introduced vars
  postConstraints( $C$ )
  CallStack.pop()
  return  $b$ 

```

root-context. If b is not known to be true, Boolean variables have to be introduced to represent c_1 and c_2 in a predicate `bool_and`(b_1, b_2), which is then flattened further.

The next case deals with the disjunction of two constraints. In this case both constraints must have control variables introduced and a predicate `bool_or`(b_1, b_2) is posted. The next case flattens predicate calls. First, the arguments to the predicate (t_1, \dots, t_n) must be flattened resulting in the set of variables v_1, \dots, v_n . If a t_j is an array, it is passed to `flata`, with the resulting array stored as the variable v_j . Otherwise, t_j is a `cons` or `term`, and a new variable v_j is introduced to store the result of flattening by `flate`. Once v_1, \dots, v_n have been instantiated, the compiler checks whether the predicate exists in *PredicateMap*. If it has a definition, it is extracted from the map and its parameters x_1, \dots, x_n are instantiated with v_1, \dots, v_n , the syntax $c_0[x_1/v_1, \dots, x_n/v_n]$ representing this instantiation. Finally, this instantiated c_0 is flattened with `flatc`. In the alternative case, where the entry for a predicate in *PredicateMap* is \perp , marking it as a builtin predicate, another conditional controls how it must be flattened. If the control variable b is true, the predicate can be

Algorithm 2.8 Flattening a term.

```

procedure flatt( $v, t$ )
   $C \leftarrow \emptyset$ 
   $CallStack.push(t)$ 
  switch  $t$ 
  case  $i$  (int):  $C \leftarrow \{v = i\}$ 
  case  $v'$  (ivar):  $unify(v, v')$ 
  case  $t_1 \oplus t_2$ :  $C \leftarrow flatt(newVar(v_1), t_1) \cup flatt(newVar(v_2), t_2) \cup$ 
     $\{flatc(true, \oplus(v_1, v_2, v))\}$ 
  case  $a[t_1]$ :  $C \leftarrow flatt(newVar(v'), t_1) \cup \{element(v', a, v)\}$ 
  case if  $c$  then  $t_1$  else  $t_2$  endif: if ( $eval(c)$ ) then  $C \leftarrow flatt(v, t_1)$ 
    else  $C \leftarrow flatt(v, t_2)$ 

   $CallStack.pop()$ 
  return  $C$ 

```

posted in the root context as $p(v_1, \dots, v_n)$. If b is not true, the postfix “_reif” is added to the predicate name and the control variable is added as the final argument. This new predicate call is then flattened using `flatc`.

The next case is that of an if statement. Here, the condition c_0 is evaluated using the function `eval`, which takes an expression that involves no variables and returns a concrete value. In this case the concrete value is a Boolean to decide which branch of the if statement should be flattened further (c_1 or c_2). The next case is for a **let** expression, where the procedure `flatlet` is used to flatten the expression’s declarations and constraint after the variables used are first replaced by renamed versions. The final case flattens `forall` loops. Here, `flata` is called with b as its first argument. Recall that, if b is known to be true, the elements of the flattened array will be flattened in the root-context. Alternatively, if the value of b is unknown, `flata` will return an array of variables $([b_1, \dots, b_n])$ which are bound to the flattened elements of the array or array comprehension. These variables are then used as arguments to an `array_bool_and` predicate which will state that b must be *true* if all of b_1, \dots, b_n are *true*. Finally, `postConstraints` is called to commit the constraints from C to the FlatZinc program, and the *CallStack* is popped.

Flattening terms. Terms are expressions that return a number or a variable representing a number. Terms are processed by the `flatt(v, t)` procedure presented in Algorithm 2.8. The procedure begins by initialising the constraint set C to be empty. Next the term is pushed onto the *CallStack*. Following this a switch statement applies the necessary case depending on the type of t . If t is an integer i , the assignment $v = i$ is added to the constraint set C . If t is another variable v' , the procedure `unify` is called. The next case is for a case involving an arithmetic expression $t_1 \oplus t_2$, where t_1 and t_2 are terms and \oplus is one of $+$, $-$, $/$, $*$ or ‘*mod*’. In this case we introduce new variables v_1 and v_2 using `newVar`. These are assigned the result of evaluating t_1 and t_2 , respectively, by recursive calls to `flatt`. The returned constraints from these calls are then added to a call to a concrete relational constraint $a(v_1, v_2, v)$, e.g., $w = x + (y + z)$ becomes the set: $\{ \text{int_plus}(y, z, v1), \text{int_plus}(x, v1, w) \}$. The next case handled by `flatt` is that of an array access $a[t_1]$, where a is an array and t_1 is a variable term. A new variable v' is introduced to take the result

of evaluating t_1 with `flatt`. An `element` constraint is then added to C : `element(v' , a , v)`. This constraint states that v must take the value from the array a indexed by v' . The last case handled by the simplified `flatt` procedure is the case of an if statement, where the condition c is not variable. The procedure will call `eval` on c and, based on the result, will flatten either the `then` branch (t_1) or the `else` branch (t_2). Finally, the *CallStack* is popped and the set C of flat constraints is returned.

Flattening let expressions. The flattening of `let` expressions is handled by the `flatlet` procedure presented in Algorithm 2.9. The procedure iterates twice through the items in d , that is, through the declarations of the `let`. The first time through, it evaluates any parameter (constant) declarations, updating d , c and t as needed. After this, it flattens any variable terms, introducing variables as needed. Finally, it returns in c the set of constraints representing the instantiation of these variables in conjunction with the updated `in` component of the `let` expression. Note that c will be flattened later.

Algorithm 2.9 Flattening a `let` expression.

```

procedure flatlet( $d, c, t$ )
  for each ( $item \in d$ ) do
    switch  $item$ 
      case  $l \dots u : v = t'$ :  $d \leftarrow d \llbracket v/eval(t') \rrbracket$ ;  $c \leftarrow c \llbracket v/eval(t') \rrbracket$ ;  $t \leftarrow t \llbracket v/eval(t') \rrbracket$ 
      case int:  $v = t'$ :  $d \leftarrow d \llbracket v/eval(t') \rrbracket$ ;  $c \leftarrow c \llbracket v/eval(t') \rrbracket$ ;  $t \leftarrow t \llbracket v/eval(t') \rrbracket$ 
      case bool:  $v = c'$ :  $d \leftarrow d \llbracket b/eval(c') \rrbracket$ ;  $c \leftarrow c \llbracket b/eval(c') \rrbracket$ ;  $t \leftarrow t \llbracket b/eval(c') \rrbracket$ 
  for each ( $item \in d$ ) do
    switch  $item$ 
      case  $l \dots u : v = t'$ :  $c \leftarrow c \cup l \leq v \cup v \leq u$ 
      case var int:  $v = t'$ :  $c \leftarrow flatt(newVar(v'), t') \cup v = v' \cup c$ 
      case var  $l \dots u : v$ :  $newVar(v')$ ;  $c \leftarrow c \cup v = v' \cup l \leq v \cup v \leq u$ 
      case var  $l \dots u : v = t'$ :  $c \leftarrow flatt(newVar(v'), t') \cup v = v' \cup$ 
         $c \cup b' \cup l \leq v \cup v \leq u$ 
      case var bool:  $b = c'$ :  $newVar(b')$ ;  $c \leftarrow c \cup (b \leftrightarrow b') \cup (b \leftrightarrow c')$ 
  return  $c$ 

```

2.5.3 Further Compilation Techniques

This section lists several techniques used by the full MiniZinc compiler that further improve the compilation of models to programs. Knowledge of these techniques is necessary to properly interpret the experimental results obtained in Chapter 4. This is because the reduced variable domains obtained by the evaluated methods increase the applicability of these techniques and, thus, the effectiveness of the compilation and the efficiency of the solving process. The techniques are not integrated into the algorithms above because a good understanding of their implementation is not necessary.

Compiling partial functions. For each subexpression e , the compiler conceptually introduces two new variables: v_e for the value of e , and a Boolean b_e representing whether e is *defined* (Stuckey et al., 2013). Consider then, the expression $e = x + (y \text{ div } z) \leq 10$.

The relational semantics (Frisch et al., 2009) dictates that this constraint holds if and only if the division is defined (i.e., $z \neq 0$) and the condition is satisfied. Flattening e will generate the following auxiliary variables:

```

v0 = div_total(y, z)      // total version of division
b0 = (z != 0)             // condition under which v0 is defined
v1 = x + v0
b1 = b0
v2 = (v1 <= 10) /\ b1     // true if defined and condition holds
b2 = true
ve = v2
be = true

```

Depending on the *context* in which the expression is translated, the algorithm will try to avoid introducing auxiliary variables and expressions as much as possible. For instance, if the above expression was stated as a top-level constraint, the value of v_e would be known to be true (since the whole constraint needs to be true). From this the algorithm can infer that v_2 and, hence, b_1 and b_0 must also be true. This forces z to be non-zero.

Overloading resolution. An important feature of MiniZinc is its support for overloading functions and predicates based on the types of their arguments. For example, in Algorithm 2.8, an array access $x[y]$ is compiled as an `element` constraint if y is a variable, but evaluated if fixed. Similarly, a solver library may declare builtin support for a `cumulative` constraint with fixed durations and capacities, but use a decomposition if those arguments are variables. When translating a call, the compiler uses the types of the *actual* arguments to decide which version to use (Nethercote et al., 2007).

Linear simplification. The linear parts of arithmetic expressions are aggregated into linear expressions, rather than being posted as sets of simpler constraints.

Boolean simplification. Any complex Boolean expression needs to be decomposed into simple constraints understood by the target solver. Before decomposing, the compiler normalises Boolean expressions by pushing negations inwards, and aggregates individual expressions into longer clauses and conjunctions. In doing so, the compiler delays the decomposition of reified constraints as long as possible to avoid reification altogether if it can be inferred that the constraint must be globally true (Feydy et al., 2011).

2.6 Model Structure

Information about the structure specified by models can be used throughout the modelling, compilation, and solving process to find solutions faster. Two general classes of structure that will be explored in the thesis and how they can be lost during compilation, or never be discovered in the first place, will be discussed. The first class is that of the *explicit structure* which relates to the particular way in which the modeller designed and

```

constraint int_lin_le([1,1,1],[i155,i243,i313],1);
constraint int_lin_le([1,1,1],[i156,i244,i314],1);
constraint int_lin_le([1,1,1],[i157,i245,i315],1);
constraint int_lin_le([1,1,1],[i158,i246,i316],1);
constraint int_lin_le([1,1,1,1,1,1],[i12,i67,i105,i159,i247,i317],1);
constraint int_lin_le([1,1,1,1,1,1],[i13,i68,i106,i160,i248,i318],1);
constraint int_lin_le([1,1,1,1,1,1],[i14,i69,i107,i161,i249,i319],1);
constraint int_lin_le([1,1,1,1,1,1],[i15,i70,i108,i162,i250,i320],1);

```

Listing 2.5: Extract golomb.fzn compiled for a MIP target.

represented the model. This includes, for example, how variables are used to represent the problem decisions, how related constraints are grouped within the model via a `forall`, and what specific constraints are used to describe the problem. The second class is that of *implicit structure* which relates to properties of the model that are not explicitly stated in it, but are entailed or implied by it. For example, a constraint of a model might be redundant, i.e., entailed by other constraints (Bessiere et al., 2007; Choi et al., 2004), a set of its variables might be symmetrically interchangeable (Gent et al., 2006b; Mears et al., 2008; Van Hentenryck et al., 2005), or a set of constraints may be semantically equivalent to a global constraint not present in the model (Leo et al., 2013).

The more structural information (both implicit and explicit) compilers and solvers are aware of, the better decisions they will make. This is why, if the information does not explicitly appear in the compiled program, some solvers will perform potentially expensive operations to rediscover it (Biere et al., 2014; Salvagnin, 2014, 2016). Therefore, retaining explicit structure and discovering implicit structure during the compilation process can be beneficial for obtaining efficient programs.

2.6.1 Losing Structure

The representation of a problem is refined during compilation from a parametric and solver independent model down to a specific set of variables and constraints for a specific solver. As the representation is refined, explicit structure might be lost. We therefore aim at reducing the amount of explicit structure that gets lost and make sure that any implicit structure we detect is explicitly represented and retained during the compilation process.

An example of this loss is the case of a `forall` loop: once it has been unrolled entirely and its specific set of constraints are in the program, there is little evidence available to the compiler (or the modeller trying to debug the output program) that these constraints are related in any way. Listing 2.5 illustrates such a case by showing a section of the FlatZinc program obtained when compiling the `golomb.mzn` model of Figure 2.4 for a MIP solver. The constraints are all the same kind of constraint (`int_lin_le`) and involve sets of introduced variables that are not explicitly mentioned in the model. It can be difficult to match these constraints and variables back to the model. Retaining knowledge of where the constraints came from, allows us to reason with more certainty about particular properties of these constraints.

Another closely related example is that of a decomposed global constraint. In the traditional compilation approach, a global constraint that is decomposed into simpler

constraints is syntactically missing from later compilation steps. The simpler constraints that make up this global are mixed in with the other compiled constraints, making it difficult to efficiently rediscover the missing high-level structure. Again, looking at the example in Listing 2.5, which shows the result of flattening for a MIP solver, we see a set of `int_lin_le` constraints. These come from the `alldifferent` constraint.

In summary, recovering lost structure from programs without the context of the original model can be difficult. Thus, it is important for the compilation process not to lose information regarding structure present in the model. A technique for achieving this is presented in Chapter 4.

2.6.2 Recovering Structure

As discussed above, sometimes the model has structures that, if made explicit, can potentially be exploited by the compiler and/or the solver. We already provided some examples of implicit structures such as redundant constraints, symmetries, and sets of constraints that are equivalent to a global constraint. Another common and practical example is that of a modeller coming from a MIP background who models their whole problem in terms of binary variables. The binary variables might be representing assignments to conceptual integer variables that do not actually occur in the model. If this implied structure (the integer variables) can be discovered and explicitly represented in the model, it may be then possible to discover global constraints on these integer variables.

Unfortunately, detecting implicit structures in a model is a complex and, thus, uncommon task. Exceptions to this include Mears et al. (2008) and Van Hentenryck et al. (2005), where symmetries among variables and values in a model are inferred. Consider the case of detecting the equivalence of a global constraint to a set of constraints in the model. As stated before, there exists many ways to model a problem and, similarly, there are also many ways to encode a global constraint. Therefore, detecting a manually written encoding of a global constraint described by the modeller in a purely syntactic way is in general a non-trivial task. While there are some trivial cases, such as a clique of dis-equality constraints representing an `alldifferent`, there are many other ways to describe this constraint that would be difficult to detect in the general case. The task is even more difficult when the modeller is unaware of the structures present in the problem. For example, when describing a scheduling subproblem in terms of linear constraints, the modeller might not realise that the conjunction of some of these constraints implies an `alldifferent` constraint on some variables. A method to detect global constraints in this general case is presented in Chapter 5.

2.7 Using Structure

The structure present in a combinatorial problem can be utilised to find solutions more efficiently. As a result, most solvers have associated *presolving* approaches that attempt to reformulate a given program (based on knowledge about its structure) in such a way as to allow the solver to find solutions faster. Sometimes the approaches require concrete data

but are useful for many different kinds of solvers and are, therefore, conceptually associated to the instance (rather than the program) level. At the level of a model there is a related concept, called *model reformulation*, where a model is analysed and reformulated to allow solutions to be found faster. This section provides a brief description of how structure can be used at the program level, at the instance level, and at the model level.

2.7.1 Program Level

This section presents a brief overview of program level presolving in LP, MIP, SAT, and CP. There is a significant amount of research on refining these kinds of programs (Andersen et al., 1995; Eén et al., 2005; Mahajan, 2010). This is because at the level of a program, concrete decisions have been made with specific instance data in mind. This yields a concrete, low-level description of a problem which admits relatively quick and simple presolving, since some simple structures can be easier to detect. A more detailed examination of presolving methods in programs and instances is presented as part of Chapter 4, which is more closely related to the topic of presolving.

Linear programming. Presolving methods have been a core part of LP solvers for decades. Most of these methods aim at tightening variable bounds and removing redundant constraints by utilising the structures represented by constraints in the program. A detailed survey of presolving methods for linear programs can be found in Andersen et al. (1995).

Mixed-integer programming. In addition to presolving for linear programs, a number of dedicated presolving techniques that target the discrete variables in a MIP program can be used (Mahajan, 2010). One important technique, called *probing* (Savelsbergh, 1994), tentatively assigns values to the zero-one variables in a program, and records the effects of the assignment. The implication graph that is constructed can expose hidden structure in a program that solvers can take advantage of when the solving begins. Probing may also fix certain variables or discover infeasibility before solving begins.

Boolean satisfiability. SAT solvers face similar challenges as LP and MIP solvers, in that their input programs can be large. This has led to the development of presolving for SAT (Eén et al., 2005), which mainly aims at fixing variables and reducing program size.

Constraint programming. While there has not been as much work done on presolving for CP as for LP or MIP, there has still been a number of studies. For example, techniques such as Singleton Arc Consistency (Bessiere et al., 2011) (or SAC) and k -Singleton Arc Consistency (Dongen, 2006) have been explored. These perform a preprocessing step similar to probing, tentatively assigning variables and removing values from domains that lead to failure before search begins.

Syntactic matching (mentioned earlier in Section 2.6.2) can be used to automatically detect particular structures in different kinds of program. While powerful, this method

is hampered by the fact that the detection algorithms are limited by the number of ways a structure can be described. A common form of syntactic matching is that of clique detection in constraint programs. The aim is to detecting cliques of simple dis-equality constraints ($<$, $>$, \neq) in a program and add the stronger `alldifferent` constraint which can improve performance considerably (Bardin et al., 2012; Frisch et al., 2003).

2.7.2 Instance Level

The previous techniques are focused on modifying a program so that it can be more efficiently solved by a particular solver. It is sometimes possible to obtain information for a program (or more accurately, an instance, since while the input data is needed, the target solver is not) that allows it to be modified in such a way as to be solved faster by all solvers of a particular class (i.e, all MIP solvers) or even any solver. This section describes these instance-level approaches.

There are two main instance-based transformations that yield good speed ups: reducing the size of the instance by eliminating variables and useless redundant constraints, or adding useful redundant information to the instance. However, it is important to note that, while having more structural information about a problem, and having less unnecessary information may sound useful, it is not always the case. The performance of a solver, given slight variations of the same instance, can be difficult to predict (Fischetti et al., 2014), with minor changes to the structure of a problem leading to significantly different performance (Liberto et al., 2016).

Reduction-based Techniques

An example of a reduction based technique designed to work at the instance, rather than at the program, level is that of Dittel et al. (2009), where the presolving performed by a MIP solver was not sufficient to allow a problem to be solved correctly. The authors deemed that the solver was not able to infer enough from their mixed-integer program and set about implementing their own problem-specific presolving which could improve an instance before producing the final program. Most notably, their approach included a technique that improved the linear representation of the problem, by using hand coded algorithms for presolving the non-linear constraints that are later implemented with linear constraints. This non-linear presolving leads to significant reductions in program size and an associated improvement in solving times, increasing the number of instances that can be solved.

Another example is that of *equi-propagation* for Boolean satisfiability (Metodi et al., 2011). The approach starts from a model in a language similar to MiniZinc that includes integer variables and constraints. It uses propagators for these constraints to help produce smaller, more efficient SAT programs.

These approaches motivate the need for presolving of higher level representations before compiling for a low-level target solver. Chapter 4 explores the reduction-based presolving of instances, by using the structure from a model to enable a CP solver to presolve an instance before compilation for a MIP solver.

Extension-based Techniques

It is often possible to extend an instance with constraints that improve the speed at which a solution can be found. The following approaches aim at finding structure in instances that can be used to construct these constraints.

Symmetry detection. Symmetry detection aims to avoid exploring branches of the search tree that correspond to symmetric (non)solutions to a problem. An example of a symmetrical solution can be found in the Golomb Ruler problem. Recall that a solution is a set of marks on the ruler. All solutions can be rearranged to find other valid solutions (e.g., $[0, 1, 4, 6]$ is the same solution as $[4, 1, 0, 6]$). As these solutions can easily be found by simple permutations of assignments to variables, it would be pointless (if trying to enumerate all solutions) to spend time looking at these symmetric solutions. More importantly, just as the valid solutions to the problem have symmetries, so do the partial assignments constructed during search that do not lead to a solution, and a lot of time can be spent exploring these redundant failures. To break this permutation symmetry we can add constraints that force the marks to be in order.

Detecting symmetries in instances has received considerable attention (Mears et al., 2009; Puget, 2005; Van Hentenryck et al., 2005). One of the most popular and accurate approaches for detecting symmetries is discussed in Puget (2005) which first creates a graph, mapping variables to the constraints they are involved in (a single vertex in the case of *alldifferent*, though different graphs are needed for different types of constraint), and then search for graph automorphisms, which correspond to symmetries. This approach was extended by Mears et al. (2009) to work for more constraint types and with several different representations of the graph. The choice of graph allows a broader range of problem types to be analysed as some representations might be significantly larger than others. A generalisation of this idea can be found in *dominance detection*, where approaches seek to prune trivially dominated branches of the search tree (Chu et al., 2012a).

Implied constraints. Implied constraints are redundant constraints that can provide a solver with an alternative view of an instance, potentially making it easier to solve.

A technique presented in Bessiere et al. (2007) finds global constraints that are implied by a subproblem of an instance. To use this technique one must start by selecting a subproblem by hand, generate many possible arguments for a global constraint and construct a large disjunction of globals with these arguments. Globals that can be easily proved to be false are then removed from the disjunction until what is left is a collection of global constraints that are implied by the subproblem. These constraints are then added to the full instance. This approach was shown to be effective for improving the solving time of some scheduling problems.

Another important related approach is that of Constraint Seeker (Beldiceanu et al., 2011), which infers global constraints from positive and negative sample arguments for a global constraint. It rearranges the arguments and matches them to global constraints found in the Global Constraint Catalog (Beldiceanu et al., 2007). It returns a ranked list

of global constraints that may fit. The constraint seeker tool is used as an intuitive way to search the catalog based on sample solutions to subproblems.

A more powerful tool that tries to not only find implied constraints, but an actual constraint program for a given set of solutions is Model Seeker from Beldiceanu et al. (2012). The Model Seeker takes a set of solutions to a problem and partitions them in various ways to construct possible arguments for a global constraint. It then submits these arguments to Constraint Seeker and tries to build a valid constraint program for the given solutions. The “model” that its title refers to is in fact an instance in our notation as it is not a parametric model and cannot be used for arbitrary data, with generalisation to a model being left as a task for the user. This approach proved effective for several well structured problems, but cannot construct constraint programs when intermediate variables are required, as these are typically not included in the solution to a problem.

2.7.3 Model Level

Changes to a model can have a drastic impact on performance, making this the most important target for improvements. While finding improvements to a model can have a big impact on solving time, there has not been as much research on model reformulation as for the presolving of instances and programs. Arguably, this is due to the fact that the model is often difficult to analyse and subsequently improve, as it is difficult to infer accurate information from the parametric constraints without data. While analysis at this level is not as common, there has been some important work which is explored here.

Symmetries

Symmetries can also be discovered and exploited at the model-level. One approach for exploiting symmetries at the model level is that of Van Hentenryck et al. (2005) in which the symmetric properties inherent in certain global constraints were exploited. The main contribution of this paper is to show that the symmetries associated with individual global constraints can be used to derive symmetries for the model via composition. This paper is also notable for suggesting some possible reformulations that make symmetry detection easier, and which could be applied as a form of automatic model reformulation.

The work of Mears et al. (2006) (later Mears et al. (2009)) for instance level symmetry detection was later used in Mears et al. (2008) to generalise the symmetries found in an instance to parametric symmetries at the model level. The approach works by taking a model and some instance data, building graphs that retain some information from the model, and then detecting symmetries in these graphs. The symmetries are combined with the identifiers from the graph to create different parameterised symmetries. The intersection of the parameterised symmetries found in all instances is a set of candidate model symmetries. These can then be filtered by using theorem proving tools or, if this is not feasible, can be presented to the user as possible model symmetries that need to be proved. The approach was shown to be effective, allowing symmetries to be detected in more complex models than was previously possible. This approach for finding structure

at the model level by classifying structures in the instances in a parametric fashion, is the inspiration behind the Globalizer approach described in Chapter 5.

Implied Constraints

As discussed in Section 2.7.2, implied constraints can improve the speed at which solutions to a problem can be found. The inference of implied constraints at the model level has not been explored as much as the inference at the instance level. An example of model level implied constraint acquisition can be found in Charnley et al. (2006a), where machine learning techniques are used to generate implied constraints for a model. The approach taken constructs implied constraints that can be proved or disproved with a theorem prover, thus restricting the types of constraints that can occur in the model and also those that can be inferred. The work restricts the types of models that can be analysed in this way to those that can be parameterised by a single integer. MiniZinc focuses on more general, realistic models that are parameterised by arbitrary input data. A method to detect implied global constraints in these more general models is presented in Chapter 5.

2.8 Conclusion

There are many techniques available that can solve combinatorial problems encoded in different ways. High-level modelling languages like MiniZinc allow users to describe a problem once at a high level of abstraction and, then, have this representation compiled, in conjunction with some instance data, into a solver-specific program. Structural information can be inferred and used to improve the modelling, compilation, and solving process supported by these modelling languages. This can be achieved by developing techniques that allow (a) explicit structure to be preserved during the compilation process, and (b) implicit structure to be discovered and made explicit. A simplified grammar and interpreter for MiniZinc was presented here, and will be built upon in later chapters, to illustrate the changes required to a compiler to implement the various techniques introduced by the thesis. A brief overview of some approaches that are used to improve programs (presolving) and higher level representations (model reformulation) was also presented. The next chapter will present a framework that allows for better reasoning about how the various techniques could interact to improve the modelling, compilation, and solving process.

Chapter 3

A Framework for Model Structure

3.1 Introduction

This chapter discusses the shortcomings of the existing modelling, compilation, and solving process explored in Section 2.5, and introduces a new framework for exploring how structure is communicated and used throughout this process. The added components of this new framework are introduced, providing a map for the remaining chapters of the thesis.

Motivation

As mentioned in Chapter 1, combinatorial problems can be modelled in many different ways, and the choice of model can have a significant impact on the efficiency of the resulting program, that is, on the time it takes to find a solution to the program. Models that make explicit their inherent structure allow users, compilers, and solvers to make better decisions during modelling, compilation, and solving. This can yield better models and programs and, as a result, faster solve times. However, this is only possible if the framework appropriately handles this structural information during compilation. Exploring the flow of structural information and how it is discovered and transformed by the different stages of the compilation process exposes ways in which the existing modelling, compilation, and solving process can be improved upon.

Contribution

The flow of structural information between the different stages of the modelling, compilation, and solving process takes three main forms. The first is the flow of structural information that occurs during the traditional translation of the high-level representation of a model to the low-level representation of a program. The second is the flow of structural information that can occur between representations at the same level (or stage) of the process, that is, *structure sharing* between several models, several instances or several programs. The third is the flow of structural information that can occur in later stages back to earlier stages in the process, that is, *structure generalisation* from a program to an instance or from an instance to a model. As we will see, both structure sharing and

structure generalisation can enable powerful improvements to the modelling, compilation, and solving process. The main contribution of this chapter is to highlight the opportunities for improving this process, by proposing a framework for exploring where structural information can be discovered and communicated. By doing so, we can develop a more efficient, productive process that allows users to produce problem models that can be solved faster. In addition, the framework provides a map that should help readers contextualise the remaining chapters of the thesis.

Structure of the Chapter

The new framework is introduced and described in Section 3.2. Section 3.3 discusses the concept of structure sharing between components at the same stage, and several examples of structure that can be found at the different stages of the framework are presented, along with how they can be shared within a stage. Section 3.4 discusses the concept of structure generalisation, where structural information is communicated back to earlier stages of the modelling, compilation, and solving process. Section 3.5 shows a diagram of the framework, that highlights the particular aspects that will be explored in the remainder of the thesis. The chapter then concludes with Section 3.6.

3.2 The New Framework

Figure 3.1 presents a view of the new framework that is proposed as an extension of the traditional modelling, compilation, and solving process presented in Figure 1.2. Recall that in the traditional process, a problem is first formalised as a model, which is then combined with data to form an instance. This instance is further specialised into an input program for a target solver, which is then used to find appropriate solutions. This traditional process is represented in the figure by the black and grey arrows going from left to right. The paths represented by the grey arrows correspond to copies of the traditional process, where the user instantiates the model with different data yielding different instances, or

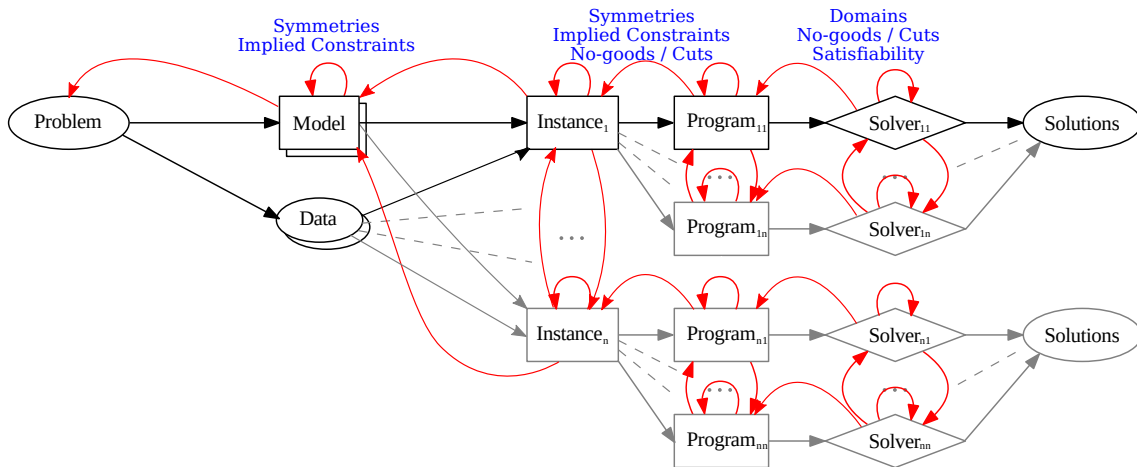


Figure 3.1: Modelling, compilation, and solving process, showing structure generalisation and structure sharing, in addition to the traditional flow of structural information.

compiles an instance for different target solvers yielding different programs. This allows us to explore the flow of information among different problem instances and programs. The iterative process of modelling and reformulating a model to produce multiple models is represented by the stack of models with red self-loops.

Augmenting the left-to-right links of the traditional process is a new set of red arrows representing opportunities for structural information to be communicated throughout the framework. In particular, vertical arrows represent opportunities available for *structure sharing*, while right-to-left arrows represent opportunities available for *structure generalisation*. The diagram also includes self-loops that represent internal sharing or exploitation of structure to improve the same representation. Some examples of structural information that can be discovered at the different stages in the process, are listed in blue text above these stages. These will be explored in more detail in the following sections.

This new framework provides a more complete view of the possible paths the process can take, and of the associated flows of structural information. Thus, it allows us to explore the greater number of opportunities this flow of information creates for improving the modelling, compilation, and solving process. Note that, by including the idea of multiple instances deriving from a model, the framework emphasises the value of good modelling, as this makes the hierarchy of model, instance, and program clearer. Changes to a model (or even more powerfully, to the modeller’s concept of the problem) will impact the quality of all instances and all programs that result from it, while an improvement to a single instance can only affect the quality of its compiled programs.

3.3 Structure Sharing

This section briefly discusses some of the types of implicit structural information that can be made explicit at different stages in the process, and how this information can be shared with other components within the same stage.

3.3.1 Between Solvers

There are certain kinds of structural information that solvers can discover relatively easily, and that would also be useful to share with other solvers working on the same problem instance. However, sharing this information is not always straightforward, as the internal representation used by solvers may be different. To illustrate this, three such kinds of implicit structural information that can be discovered will be presented, and the issues of sharing these with other solvers working on the same problem instance will be discussed.

The first kind of information is one of the simplest a solver can infer and share: an answer to whether or not the program being solved is satisfiable. There are several uses for this kind of information, beyond just learning whether the instance is satisfiable or not. For example, when trying to debug an unsatisfiable instance of a problem, it can be useful to learn which subsets of constraints cause unsatisfiability. Most approaches for this rely on the solver to get this information, and do so by repeatedly calling it with different subsets of program constraints (Bailey et al., 2005; Liffiton et al., 2015). Another example

of the use of this kind of information is the case of MIP solvers, where an LP solver is used to solve subproblems derived from the input program during search. Discovery of an infeasible subproblem is communicated back to the MIP solver to ensure the correct action can be taken. Communicating this is relatively simple, as it only requires passing a “satisfiable” or “unsatisfiable” (or 1/0) message.

A second kind of information that can be inferred and shared among solvers working on the same instance is nogoods. This is useful, for example, to improve performance when solving a program with a portfolio of different configurations of the same SAT solver (Audemard et al., 2014). Communicating this kind of information is possible because the internal representations of these solvers should be similar and, therefore, nogoods involving literals from one instance can be easily added to another. However, if the internal encodings of the program are different, specialised algorithms for translating a nogood must be used.

The third kind of information discussed here is that of variable domains. This is useful, for example, for hybrid solvers, which try to benefit from the strengths of two different solving technologies. They do this by sharing implicit information about an instance that is learned during the solving of a program (Bockmayr et al., 2006; Yunes et al., 2010). This can be achieved by, for example, using a CP solver to propagate complex global constraints with dedicated algorithms that outperform the general inference possible with a decomposition given to a MIP solver. Sharing this information between solvers can be difficult when their internal representations are different (which is often the case). This is why solvers usually only share the domains of a few variables, in many cases just the bounds on the objective. To increase the opportunities for sharing, either a custom translation must be implemented to share information between different representations of a problem, or both solvers are given an identical – lowest-common denominator – representation, which will make the inferences for one technology weaker but allow more to be shared. For example, a MIP formulation may be presented to both the CP solver and the MIP solver, as the formulation can be understood by both.

3.3.2 Between Programs

As shown in the framework, each instance can yield several possible programs depending on the target solver. Since the programs come from the same instance, any inference made about any one of them is true for all of them. There are certain kinds of structural information that can be easier to infer at the program level than at the instance level, as the exact variables and constraints that will be used by the solver are known at the program (but not at the instance) level. This section discusses what information can be discovered in a program and shared with other programs derived from the same instance.

As explored in Section 2.7, symmetries are an example of structural information that can be exposed at the program level (Mears et al., 2008). While it is correct for symmetries to be communicated between programs (as they are based on the same problem data), it is not always easy. Different programs may present considerably different views of an instance, making it difficult to match similar components of different programs. For

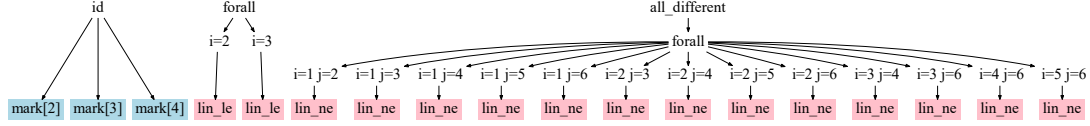


Figure 3.2: Trace of compilation of `golomb.mzn` (Figure 2.4) with $m=4$.

example, a symmetry in a set of variables introduced in one program may be impossible to match to another set without a complex translation step.

Another kind of information that can be inferred from programs is that of implied constraints (Bessiere et al., 2005b; Charnley et al., 2006a; Frisch et al., 2003). One way to infer implied constraints is to simply examine what constraints are explicitly present in the program. This approach is called syntactic matching (Frisch et al., 2003). One example is the search for cliques of constraints that imply dis-equality between variables. A clique like this can be replaced or accompanied by an `alldifferent` constraint, which can improve solving performance. It could thus be advantageous to share the discovery of such structure with other programs. However, a difficulty arises when sharing with programs specialised for a solver that does not support global constraints. At the program level the instance has already been compiled for a specific solver and, therefore, the constraint may not be supported. In fact, the variables that the implied constraint is related to may not even be present in the target program in some cases. As such, it is clear that to gain the most from this inferred information, it must be presented at the higher level of instance.

As seen above, sharing non-trivial information at the solver level and at the program level requires mapping components of one program to another. To this end, we introduce the concept of *variable paths* and *constraint paths*, or, collectively, MiniZinc paths. These paths can provide unique identifiers for concrete items at the program-level expressed in terms of the model. The approach records a trace of how a model and its data is compiled, recording paths that describe why variables and constraints were introduced. A simplified trace of the paths that lead to the introduction of variables and constraints for a compilation of the `golomb.mzn` model with $m=4$ from Chapter 2 is presented in Figure 3.2. From this diagram we can see the introduction of the three `mark` variables (`mark[1]` being a fixed to 0), the two `int_le` constraints added by a `forall`, and also the set of `lin_ne` constraints introduced by the `all_different` constraint. Note how the trace includes information regarding which loop iterations introduced each constraint. Matching common parts of different compilations of an instance is made easier by this representation. This is described in more detail in Chapter 4.

3.3.3 Between Instances

Sharing structure learned from one instance with another can also be beneficial. It is, however, more difficult than sharing at later stages, as the explicit and implicit structures found in one instance, being dependent on the data, may not be present in the next.

An example of information that can be shared between instances can be found in Chu et al. (2012b), where the authors present a method for sharing the nogoods learned from one instance of a problem with other instances. They present two approaches for achieving this.

The first is to treat the parameters of a model as variables and, rather than compiling each instance into a separate specialised program, compile once without any data (the model no longer has “parameters” in the MiniZinc sense) and fix these parameter variables after compiling to a program at the solver level. This essentially allows the nogoods to include these parameters, which would otherwise have been removed by compilation. One of the major downsides of this is that the compilation will have to use the most general versions of some constraints. For example, constraint $p[1]*v[1] + p[2]*v[2] \leq 10$, where p is a parameter represented as a variable, must be compiled as two multiplications and a linear constraint on the results (stored in introduced variables), while given actual instance data, it would be compiled as a single linear constraint. Alternatively, and more flexibly, they present a method for allowing the solver to produce “parameterised nogoods”, which are expressed in terms of the model parameters. This allows later instances to be compiled separately in the traditional way, while still allowing the solver to use these nogoods.

Communicating symmetries described at the instance level to the lower stages of compilation is relatively easy, as they are expressed in terms of the variables and constraints that will exist in some form in all resulting programs. The compiler can then make decisions about whether it is useful to keep the associated symmetry breaking constraints in the program or not (e.g., solvers that use local search methods may choose to ignore symmetry breaking constraints as they can hurt performance). However, sharing information regarding symmetries inferred for a given instance with other instances of the same model is harder to achieve as, depending on the type of problem, the input data can lead to considerably different instances. Therefore, there will typically not be enough information available within the modelling, compilation, and solving process to prove that a symmetry discovered for one instance is valid for another.

Similarly, the implied constraints described at the instance level can be communicated safely and easily down to the lower levels, offering the compiler more freedom in how it compiles an instance to a program for a specific solver. However, again, sharing these constraints with other instances is difficult, as one needs to prove that they are in fact valid for them too.

As in the case of the parameterised nogoods, the sharing of symmetries and implied constraints between instances can be achieved by describing the information in terms of the parametric model. Some of the challenges and solutions for this are presented in the next section about structure generalisation.

3.4 Structure Generalisation

Possible opportunities to generalise structural information are represented in Figure 3.1 by the red right-to-left links. Generalisation seeks to introduce higher level structure for the framework to use, that is, it generalises the structure inferred at lower levels, making it available earlier in the framework for later compilations. This section briefly discusses the different approaches proposed to achieve this at each stage of the process.

3.4.1 From Solvers

To enable information sharing at the level of individual solvers, we can attempt to generalise the structural information to the level of a program. Generalising structural information from the solver-level can be challenging, as different solvers have different internal representations of problems. A solution, then, is for the solver to express the learned structural information explicitly in terms of the program and deal with the generalisation at this level. This has the added advantage of avoiding the need for specialised handlers to be implemented.

3.4.2 From Programs

To enable information sharing at the level of a program, we can generalise the learned information to the level of an instance. Generalising the information found in programs to their respective instances would allow all future programs generated from these same instances to benefit (e.g., in portfolio solving or multi-pass compilation). This can be achieved by using MiniZinc paths. These paths explicitly represent the structure of an instantiated model and the path the compiler took through translation from model to instance to program. Once the information is available at the instance level, all future compilations of the same instance can benefit from what was learned.

3.4.3 From Instances

Presenting the structure found in instances of a problem in terms of a model can be difficult, as what is true for an instance is not necessarily true for a model. The approach taken in this thesis for generalising to the model level, inspired by Mears et al. (2008), allows us to generalise found structures to a model based on a subset of instances. The full set of possible instances for a model is referred to as the *parameter space* of that model. Without total knowledge of the parameter space it is difficult to prove that structural information discovered about one or several instances is, in fact, going to be present in all instances. However, if a discovered structure occurs across a set of instances, the structure is a *likely* candidate that can be presented as such to the user.

This thesis presents two approaches for generalising information to the model level. The first, referred to as *sample based generalisation* and explored in Chapter 5, constructs candidate structures based on the contents of the model. The presence of these candidates is then checked in several instances. If they are present in all of them, they can be considered likely candidates that can be presented to the user. Since the candidates are already described in terms of the model and use variables and arrays that exist at that level, they are not tied to a specific instance and can be passed to all. The second approach, referred to as *path based generalisation* and explored in Chapter 6, is to record the MiniZinc paths of the variables and constraints created during compilation. Then, when structures are found at the program or instance level, the instance specific components of the paths are removed so that they can be matched.

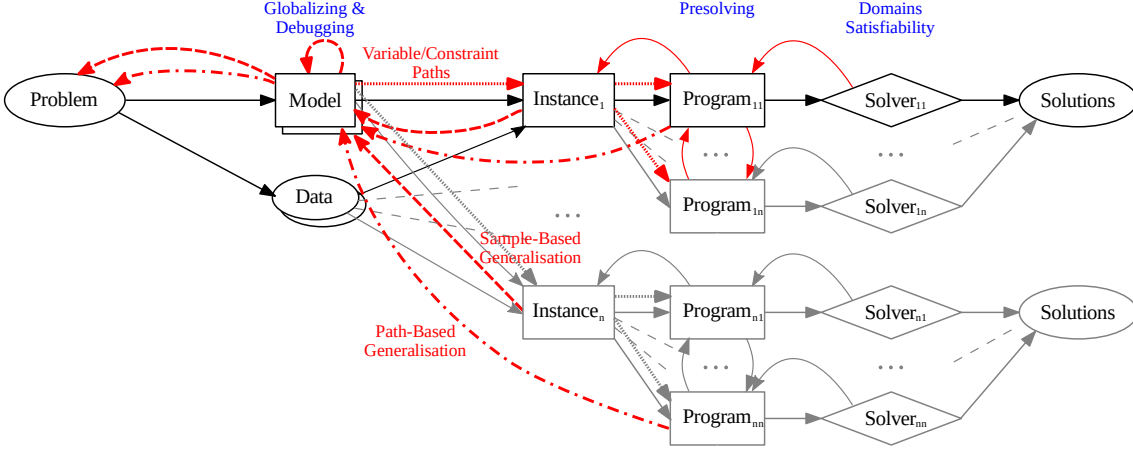


Figure 3.3: Modelling, compilation, and solving process with MiniZinc paths and instance generalisation shown.

As an example, consider a model with a set of dis-equality constraints between the variables in an array called x . The sample based generalisation approach constructs candidate constraints using explicit structure present in the model such as, for example, an `alldifferent(x)`. It then tests whether this constraint is valid by instantiating it with different instances and testing its presence. If it is present in all instances tested, the `alldifferent(x)` constraint is marked as a likely candidate and presented to the user. Instead, the path based generalisation approach compiles each instance into a program, each of which is then used to detect structural information such as cliques of dis-equality constraints. Since this approach works at the program level, the inferred `alldifferent` constraints will involve sets of variables with different names and arity (e.g., $\{x_1, x_2, x_3\}$ and $\{x_1, x_2, x_3, x_4\}$). To match these variables, we must compare the paths that uniquely identify each variable. The variable paths will only differ in the indices used when introducing them: $i = 1$, $i = 2$, etc. Replacing these data-dependent indexes with a placeholder, would make the set of paths identical if they are all from the same array. This *parameterised* path must be identical in every instance tested for this structure to be considered a likely candidate.

3.5 Additions to the Framework

Figure 3.3 presents a view of the particular elements of the framework that later chapters will explore. The blue headings represent the different goals that are explored in the thesis. At the model level, the globalizing of constraint models and the debugging of unsatisfiable models is explored. At the program level, the debugging of a single instance and multi-pass presolving are explored. Finally, at the level of a solver, domains and satisfiability information is discovered and used to achieve these other goals. The dotted red lines going from left to right show the representation of explicit structure contained in variable and constraint paths, being passed down from a model to an instance and then a program. This preserved explicit representation of the structure is used in Chapters 4 and 6 to identify parts of programs and to enable the generalisation of inferred information from the program back to the instance and model. The dot-dashed and dashed red lines

represent the generalisation techniques used in Chapters 5 and 6 respectively. These allow the generalised information learned from multiple instances of the same model to iteratively improve a model and potentially improve the user's understanding of the problem itself (exposing implicit structure that may not have been evident, or exposing mistakes in the problem specification).

3.6 Conclusion

This chapter introduced a new framework for exploring the flow of structural information during the modelling, compilation, and solving process. This new framework extends the traditional framework by introducing two additional flows of information related to the sharing of structural information between components at the same stage of compilation, and to the generalisation of structural information from later compilation stages to earlier ones. These two flows allow for a more flexible and powerful process. With improved sharing, the strengths of different solving techniques can be better combined to find solutions to an instance faster. With generalisation, the structures learned in several instances can be presented to a user in terms of the original model, thus allowing the user to improve their model.

Chapter 4

Preserving Structure

4.1 Introduction

In Chapter 2 the concept of *presolving* was briefly introduced as one of the ways in which explicit low-level structure available in a program can be used to modify the formulation of the program to improve solve times. Presolving is commonly used in linear and mixed-integer linear programming (LP/MIP) solvers and in Boolean satisfiability (SAT) solvers. However, these solvers cannot easily exploit most of the *explicit structure* available in the model, as it is often lost during the compilation process. This chapter answers the sub-goal presented in Chapter 1:

Can the explicit structure in models be used to improve the quality of programs compiled from them?

To achieve this, it first presents a method that allows the structure present in a model to be preserved further into the compilation process than before. It then demonstrates how this structure can be used to improve the quality of the programs produced by a compiler for a modelling language.

Motivation

Traditional presolving methods analyse and improve a constraint program before the actual solving process starts, inferring tighter variable domains, simplifying constraints, and removing variables and constraints that are guaranteed not to contribute to a solution. Section 4.2 presents some of these techniques in more detail. A compiler for a high-level constraint modelling language performs similar optimisations when it translates a constraint model to a solver-level program. It will try to compute tight variable bounds, select the most suitable variant of each constraint, and generate a compact program that does not contain unnecessary variables or constraints.

The main difference between a presolver and an optimising compiler for a constraint modelling language is the level of detail available for analysis. A presolver can analyse the entire compiled program, with all data, variables, and constraints present and accessible. In traditional compiler terminology, this would be called *whole program optimisation*. A compiler for a modelling language, on the other hand, typically performs the analysis and

compilation simultaneously, while it is constructing the solver-level program. Therefore, the compiler’s knowledge of the whole generated program is, inevitably, only partial until it is finished compiling. However, it can use the structure available in the input model, such as global constraints, functional dependencies, and a rich expression language, to perform more powerful optimisations.

The above situation motivates the development of techniques for *integrating whole-program optimisation* into compilers for constraint modelling languages. The resulting system should combine the advantages of whole program optimisation with the additional inference strength possible due to the model structure being available. This can be achieved by *preserving structure* and making use of this structure during *multiple compilation passes*. The initial passes would translate the input model into representations suitable for whole-program analysis, preserving other representations of model structure for use in later passes. A final pass would use the information gained from the earlier analysis, e.g., tightened variable domains and learned variable aliases, and combines it with the preserved structure to produce efficient programs.

Contribution

This chapter presents two main technical contributions. The first contribution is a method that **preserves structure deep into the compilation process in the form of a novel variable synchronisation method**. This method works by computing unique variable identifiers called *variable paths* that are stable across different compilations. The second contribution is an approach for **multi-pass presolving that makes use of the model structure represented by variable paths** to integrate whole program optimisation. We refer to this approach as path-based multi-pass presolving. This method to preserve structure has a wide range of important applications in the analysis, compilation, solving, and debugging of constraint problems. This alone can provide benefits to the compilation as the compiler implements some simple propagation which can allow for better decisions to be made on a second pass. More powerfully, the path-based multi-pass approach enables the compiler to generate a better program for the target solver (e.g., a MIP solver) by first compiling an intermediate program more suitable for presolving in a different solver (e.g., a CP solver). Using the information learned from presolving on that program, in conjunction with the variable paths, allows the compiler to then generate a better program for the target solver. The power of this approach is illustrated by using the strong inference methods of a constraint propagation solver to strengthen the compilation of models to a target solver that is not propagation-based. The experiments show that a path-based two pass compilation with a propagation based presolving pass, can produce significantly better solver-level programs, reduce the numbers of variables and constraints, tighten variable bounds, and lead to faster solving.

Structure of the Chapter

Section 4.2 presents a summary of the most common presolving methods currently applied in LP, MIP, SAT, and CP, as well as related work regarding multi-pass compilation.

Section 4.3 introduces the concept of variable-paths and their use in the path-based multi-pass presolving approach. The actual implementation of this approach for the MiniZinc modelling language is explored in Section 4.4. Section 4.5 presents experimental results that show the new technique is efficient and effective. Section 4.5.3 presents a case study showing how the path-based multi-pass approach can be beneficial for a specific problem class. Finally, Section 4.6 presents the conclusions.

4.2 Background

The concept of presolving was presented in Section 2.7. This section presents a brief overview of the existing presolving methods in LP, MIP, SAT and CP. After this, it discusses related work regarding multi-pass optimisation.

4.2.1 Presolving Linear Programs

Presolving originated in the field of Linear Programming (LP) and has been a core part of any LP solver for decades (Andersen et al., 1995). Most LP presolving methods aim at tightening variable bounds and removing redundant constraints. To simplify the discussion, this section will use a formulation of linear programs involving only equality constraints. Note, however, that there is no loss of generality, as inequalities can be easily implemented by introducing slack variables. Consider the following characterisation of a linear program:

$$\begin{array}{ll}
 \text{minimize:} & \sum_{j \in C} c_j x_j \\
 \text{subject to:} & \sum_{j \in C} a_{ij} x_j = b_i \quad \forall i=1^m \\
 & l_j \leq x_j \leq u_j \quad j \in N \\
 & x_j \in \mathbb{R} \quad j \in C
 \end{array}$$

which is identical to that presented in Section 2.3.1, except it is restricted to equality constraints. It has been found that reducing the size of the coefficient matrix a often results in quicker solve times (Gondzio, 1997). Therefore, presolving for LP aims to minimise the number of variables (columns), constraints (rows) and nonzeros (coefficients) in the constraint matrix a , and to maximise the number of linearly independent constraints. Further, presolving can sometimes discover solutions or infeasibility without needing to perform any search.

Presolving is applied repeatedly to a linear program until the reductions are too minor to waste time on. Most of the presolving techniques applied in each iteration are relatively easy to perform, as they focus on the detection of quite simple structures. Nevertheless, they can yield a large reduction in program size that allows for large improvements in

solving time. Further, these techniques can also be applied to the dual of a linear program (that is, an alternative view of the program that can be used to find an upper bound on the objective value), yielding further information. The following is a brief summary (based on the survey of Andersen et al. (1995)) of the most common structures exploited in LP presolving.

Infeasible variables: A variable is infeasible if its lower bound is greater than its upper bound. These variables indicate an infeasible instance or possibly an incorrect model.

Fixed variables: A variable x_j is fixed if its lower bound l_j is equal to its upper bound u_j . These variables can be substituted out of the program by modifying the bound b_i : $\forall_i b_i \leftarrow b_i - a_{ij}l_j$. One way to detect fixed variables is to look for rows involving only a single variable. These are called singleton rows. The fixed value can be calculated simply as $x_j = b_i/a_{ij}$ where i is the index of the singleton row and x_j is the variable to be fixed.

Free variables: A variable x_j is free if its value is unbounded. That is, if $l_j = \infty$. The values of these variables can be assigned after assigning the other variables.

Column singletons: A column j is a singleton if the coefficients of x_j are 0 in all rows except one. This indicates a variable that appears in a single constraint (row i). Techniques can improve a linear program by making use of column singletons as follows:

- Given a free column singleton j , that is, if x_j is a free variable, then x_j can be fixed as follows:

$$x_j = \frac{b_i - \sum_{p \neq j} a_{ip}x_p}{a_{ij}}$$

- If the column singleton occurs in a constraint with only one other variable (a two variable constraint), the bounds on a row can be modified so that the feasible region is unchanged when bounds on the singleton variable are removed.
- When a column singleton is discovered, a check can be performed to find whether it is “implied free” by calculating bounds on the variable. If the bounds are at least as tight as the original ones, the variable can be treated as a free column singleton.

Forcing, dominated, and infeasible constraints: Finding the minimum and maximum value that a constraint can sum to in isolation (a simple operation), allows bounds to be tightened and can fix variables if one of the calculated bounds is equal to both the upper and lower bounds of a constraint (an equation). Finding forcing and infeasible constraints works as follows: Let $P_i = \{j | \forall_{ij}, a_{ij} > 0\}$, be the set of variables with only positive coefficients, and $N_i = \{j | \forall_{ij}, a_{ij} < 0\}$, be the set of variables with only negative coefficients. Lower and upper row sums L_i and U_i can be computed for each row i as: $L_i = \sum_{j \in P_i} a_{ij}l_j + \sum_{j \in N_i} a_{ij}u_j$ and $U_i = \sum_{j \in P_i} a_{ij}u_j + \sum_{j \in N_i} a_{ij}l_j$ where P and N are columns with positive and negative coefficients respectively.

- If $\exists_i : L_i > b_i \vee U_i < b_i$, the constraint i is infeasible.
- If $\exists_i : L_i = b_i \vee U_i = b_i$, the constraint i is a forcing constraint and variables in this row can be assigned to their bounds depending on their objective coefficient.

These techniques can also be applied to the dual of the linear program to find other infeasible and forcing columns.

Strengthening constraint bounds: Savelsbergh (1994) presents techniques for strengthening (tightening) variable bounds and removing redundant constraints. If bu_i and L_i are finite, then tighter bounds l'_j and u'_j can be computed as follows:

$$u'_j = \text{Min}\{l_j + \frac{(bu_i - L_i)}{a_{ij}}, u_j\} \text{ for } j \in P_i$$

$$l'_j = \text{Max}\{u_j + \frac{(bu_i - L_i)}{a_{ij}}, l_j\} \text{ for } j \in N_i$$

Similarly, if bl_i and U_i are finite then:

$$u'_j = \text{Min}\{l_j + \frac{(bl_i - U_i)}{a_{ij}}, u_j\} \text{ for } j \in N_i$$

$$l'_j = \text{Max}\{u_j + \frac{(bl_i - U_i)}{a_{ij}}, l_j\} \text{ for } j \in P_i$$

For integer variables the upper and lower bounds should be rounded to integral values.

Further, A constraint i is redundant if bu_i is not finite and $L_i > bl_i$. Similarly for $U_i < bu_i$ if bl_i is not finite. These can be removed from the program.

Variable locks: Up-locks and down-locks are a measure of how often a variable is bound from a particular direction (Achterberg, 2009; Heinz et al., 2013). If a variable is only bound from a single direction (counting its coefficient in the objective function as a bound) it can be fixed to one of its bounds without affecting the optimality of the solution. For example, if a variable x is only locked from above it can be fixed to its lower bound.

Conclusion

LP presolving is simple and powerful owing to the fact that the language of linear constraints is simple and uniform. Note that many of the techniques used for presolving in LP aim at finding and removing redundant parts of the formulation. This hints to weaknesses in the way these programs are constructed. In fact, the typical modelling languages for these kinds of problems directly encode the users' constraints with little analysis. It stands to reason that analysing an instance before it is completely compiled to a program can save the solver from having to perform much of the presolving.

4.2.2 Presolving Mixed Integer Programs

Since MIP uses the same type of matrix as LP, the LP presolving methods detailed above can also be applied here. In addition, MIP presolvers take advantage of the integer variables present in the MIP, and may even apply presolving that specifically focuses on the relationships between assignments to binary variables. Presolving for MIP programs has been explored in many different works including Crowder et al. (1983), Dittel et al. (2009), Savelsbergh (1994), and Suhl et al. (1994). For MIP programs, reducing the size of the matrix does not always improve solving time. While the solve time for individual LP subproblems can be improved, the number of linear programs that need to be solved may increase in a non-trivial way. In practice, presolving for MIP is generally considered to be beneficial, although it is also quite unpredictable, since MIP solvers can be sensitive to even small changes in input (Fischetti et al., 2014). The following briefly summarises some of the most common presolving techniques used in MIP.

Row Classification

In (Hoffman et al., 1991) a presolve procedure is described that classifies rows in the constraint matrix and performs different presolving for the different types of constraint. The different classes of constraints described are:

SOS and Invariant Knapsack: Constraints containing only zeroes, ones, and minus ones are either *special-ordered sets* or *invariant knapsack*. If the constraint is of the form $\sum_{j \in P_i} x_j + \sum_{j \in N_i} x_j \leq 1 - |N_i|$, it is classified as a special-ordered set. Otherwise, it is an invariant knapsack.

Plant Location and Reverse Plant Location: If the constraint is of the form

$\sum_{j \in P_i \cup N_i} x_j \leq M x_k$ the constraint is a plant-location constraint. Note that if M is greater than $|P_i \cup N_i|$, coefficient reduction (see below) will reduce it to $|P_i \cup N_i|$.

If the constraint has the same form as an SOS constraint, except the expression is $\geq 1 - |N_i|$ instead, it is classified as a “reverse plant location” constraint.

Knapsack: All other constraints are considered knapsack constraints.

The above classes of constraints are then used by various presolving techniques, including: Euclidean reduction and coefficient reduction. Euclidean reduction can be performed on any knapsack constraint with rational coefficients to discover the smallest number k (sometimes called *KEXP* in the literature) such that all coefficients multiplied by 10^k are discrete. The Euclidean reduction method finds this number and the greatest common denominator (GCD) of the resulting coefficients. The constraint is then multiplied by 10^k and divided by the GCD. If the scaled constraint is an equation and has a non-integer bound, the problem has no feasible integer solution. If the constraint is an inequality, the bound is rounded to an integer value (\leq round down, \geq round up). Following this, the minimum coefficient is subtracted from the maximum coefficient in the constraint and, if it is equal to 0, the constraint is reclassified as an SOS constraint.

Coefficient reduction tightens the LP relaxation by reducing the size of the coefficients of the constraints. Transforming a constraint of the form: $\dots \geq b_i$ all coefficients $> b_i$ can be adjusted to be equal to b_i when all variables are binary. More powerful coefficient reduction can be achieved by combining SOS constraints that overlap with a knapsack constraint to tighten the coefficients on the knapsack constraint. Plant location constraints can also be used to tighten the coefficients of a knapsack constraint. This can be achieved as follows. Given a knapsack constraint A_i , find a plant location constraint that shares the plant variable x_p and at least one more variable. Create a new constraint, where K is the index set of the knapsack constraint and P is the index set of the plant location constraint without the plant variable:

$$\sum_{j \in K \setminus P} a_{ij}x_j + \sum_{j \in K \cap P} a_{ij}x_j + a_{ip}x_p \leq b_i$$

If we solve, where all $a_{ij} > 0$:

$$d = \max\left\{ \sum_{j \in K \setminus P} a_{ij}x_j \mid \sum_{j \in K \setminus P} a_{ij}x_j \leq b_i \right\}$$

we can make the substitutions $a_{ip} = b_i - d$ and $b_i = d$.

Additionally, set partitioning, covering, and packing inequalities are mentioned in Mahajan (2010) as a few more classes of constraint that can be detected and exploited.

Infeasibility and Simple Redundancy

This technique looks for two identical rows in the coefficient matrix a_{i_1} and a_{i_2} , so that:

- If both constraints are equations and $b_{i_1} = b_{i_2}$, it can remove one of the constraints, otherwise the problem is infeasible.
- If one of the constraints is an inequality and the other is an equation and $b_{i_1} = b_{i_2}$, it can remove the inequality. If $b_{i_1} \neq b_{i_2}$ the problem is infeasible.
- If both constraints are inequalities, they either form an equation or one is redundant.

Probing

Probing is used in Savelsbergh (1994) and Suhl et al. (1994) to explore the effect that assigning some binary variable has on the problem. Probing is performed by assigning binary variables to either 0 or 1 and analysing the resulting program. The following can be inferred from probing:

Infeasibility: If infeasibility is detected, the variable can be fixed to the opposite assignment.

Fixing other variables: If setting a variable x_j to both 0 and 1 result in a variable x_k being assigned to a certain value v , x_k can be fixed to v and removed.

Implication Graph: Presolving a model can produce useful implications between variables. For example, if probing the assignment $x_{j_1} = 1 \wedge x_{j_2} = 1$ results in an infeasible model, we can identify the implications $x_{j_1} = 1 \Rightarrow x_{j_2} = 0$ and $x_{j_2} = 1 \Rightarrow x_{j_1} = 0$. These implications can be useful for strengthening solving.

Cliques: Cliques of incompatible assignments can be discovered in the implication graph allowing for the construction of “clique cuts”. These cliques can be discovered by building a graph of literals (assignments to variables or to their negation, i.e., $\bar{x} = 0$), where edges between literals indicate that they cannot be assigned the same value at the same time. Cliques can be used to create *clique cuts* that can be added to the model. Larger cliques provide more useful cuts (Suhl et al., 1994). With a clique $C = C^o \cup C^c$, where C^o is a set of original variables in the clique and C^c is a set of negated variables in the clique, the following can be deduced:

- If $|C^o \cap C^c| = 1$ and $k \in C^o \cap C^c$, then $x_j = 0$ for all $j \in C^o \setminus \{k\}$ and $x_j = 1$ for all $j \in C^c \setminus \{k\}$
- If $|C^o \cap C^c| > 1$ the problem is infeasible
- The following cut can be added: $\sum_{j \in C^o} x_j - \sum_{j \in C^c} x_j \leq 1 - |C^c|$

Implication Inequalities: Logical implications can also be used for creating useful cuts. For example, $x_i = 1 \Rightarrow y_j = v_j$ implies $y_i \geq l_j + (v_j - l_j)x_i$, where x_i is a binary variable, y_j is any variable, and v_j is some value to which y_j is fixed during probing. These can provide stronger linear approximations of the feasible region than the original constraints (Suhl et al., 1994).

Detecting Permutation Structures

An approach for detecting and exploiting permutation structures (such as `alldifferent`) is presented in Salvagnin (2014). The approach first attempts to find the constraints in a mixed integer program that encode what is called an assignment structure. Once this structure has been found, the algorithm checks whether an assignment to the variables in the structure can allow the remaining variables in the problem to be trivially assigned. If so, this can then be exploited by a heuristic that focuses on just the variables in the assignment structure, in this case, a specialised local search heuristic. This highlights again the importance of high-level modelling, as these structures can be explicitly represented in a model, avoiding the need for this rediscovery entirely.

Detecting Semantic Groups

Another example of structure that can be rediscovered from a MIP program is that of semantic groups. Typically a model for a problem will include several distinct sets of variables that represent different aspects of a problem. The grouping of variables is explicit in the model but once compiled for a program the variables are separated into individual integer and binary variables. It is possible for the semantic grouping of variables and constraints to be exploited by a MIP solver to improve branching and other heuristics.

Without the detection of semantic groups, a solver simply applies different heuristics depending on the type of a variable or constraint. Salvagnin (2016) describes approaches for detecting semantic groups using partition refinement techniques where the variables and constraints of a problem are partitioned into initial groupings which are then iteratively refined into smaller groups. These techniques are shown to effectively recover semantic groups in experiments.

Conclusion

When it comes to MIP, more complex presolving techniques can be applied than (and in addition to) those used in LP. Techniques such as probing and the detection of various kinds of higher level constraints (such as permutation structures and semantic groups), highlight the fact that some of the structure that the solver can exploit while solving a MIP program, is lost during compilation and must be rediscovered. This, again, motivates the need for presolving that analyses a higher-level representation of a problem, where these structures are still present.

4.2.3 Presolving Boolean Satisfiability

As described in Section 2.3.3, a SAT program is described as a conjunction of clauses C , where each clause $c_i \in C$ is a set of disjunct literals, and each literal represents a variable x or its negation $\neg x$. The following briefly summarises some of the most common presolving techniques used to simplify SAT programs.

Unit propagation. This is the prime propagation mechanism of SAT. Given any clause that contains a single literal (e.g., x), propagation removes all clauses that include x (as they are trivially true), and removes its negation (e.g., $\neg x$) from all other clauses.

Pure literal rule. If a variable only occurs in a particular polarity, it can be fixed to this polarity. For example, if the variable x only ever occurs as the literal $\neg x$, it can be fixed to *false* (Eén et al., 2005).

Clause distribution. Given two clauses $\{x, a_1, \dots, a_n\}$ and $\{\neg x, b_1, \dots, b_m\}$ the implied clause $\{a_1, \dots, a_n, b_1, \dots, b_m\}$ is called the resolvent. A program can be simplified by first aggregating all clauses involving some variable x and its negation $\neg x$ into S_x and $S_{\neg x}$, respectively, and then pairwise resolving the clauses of S_x and $S_{\neg x}$. All non-trivial clauses are kept. To reduce the cost of performing clause distribution, it can be restricted to clauses of ≤ 3 literals as in Satz (Li et al., 1997).

Equivalent literals in the order encoding. Recall the order encoding from Section 2.3.3. Removing a value from the domain of a variable represented with the order encoding is achieved by equating some of the literals. For example, given an integer variable $0 \leq X \leq 9$, represented by 9 Boolean variables $X = [x_1, \dots, x_9]$, it is possible to specify that X cannot take a specific value v by equating $x_v = x_{v+1}$ (Metodi et al., 2011). Assume

X cannot take one of the values in $\{2, 5, 7\}$. We can equate some of the SAT variables representing X as follows: $X = [x_1, \underline{x_2}, \underline{x_2}, x_4, \underline{x_5}, \underline{x_5}, x_7, x_7, x_9]$, where underlining highlights the variables that are now unified. In this way, variables x_3 , x_6 , and x_8 have been removed from the representation, allowing a reduction in the number of variables in the CNF.

Clause subsumption. A clause c_1 is said to subsume c_2 , if $c_1 \subseteq c_2$. A subsumed clause may be removed from the problem as it is redundant.

Detecting structures in SAT formulations. In Biere et al. (2014) two approaches for detecting cardinality constraints (e.g., $x_1 + x_2 + \neg x_3 + \neg x_4 \leq 2$) in SAT encodings of problems are presented. The first approach uses explicit syntactic structure to find **at-most-one** and **at-most-two** constraints. The second uses a semantic method to find implicit **at-most- k** constraints, generating candidate cardinality constraints and testing whether they are present or not. Detecting and making these constraints explicit, can allow SMT solvers to better solve CNF programs. This again motivates the need to use a high-level modelling language and preserve the structural information during compilation, rather than writing the CNF directly, as these higher-level constraints should not have to be rediscovered.

Conclusion

Presolving has been found to be important for solving SAT, with current techniques relying on the simple language of CNF. Again, we see that some problem structure is being rediscovered during the presolving process, as in the case of Biere et al. (2014).

4.2.4 Presolving Constraint Programs

The most common presolving techniques in CP aim at reducing domain sizes, tighten bounds and remove redundant parts of a constraint program. Note that in CP, reducing the size of a model does not necessarily result in faster solve times as, for example, redundant constraints can help propagation by providing an alternative view of the relationship between variables in the instance. Ideally, a constraint should only be removed if it is both propagation and performance redundant, that is, it does not duplicate inference already available in the model and it can actually make the instance easier to solve (Choi et al., 2004). However, this information is not always easy to deduce.

Common subexpression elimination. This is a common compiler optimization where repeated expressions can be removed by creating a new variable that is equal to the expression, and using this new variable in place of the expression throughout the rest of the model (Rendl et al., 2009). This has been shown to often lead to a significant improvement in solving time in CP. The introduction of functions to MiniZinc (Stuckey et al., 2013) brought with it an implementation of common subexpression elimination, which can almost halve the number of constraints needed to describe some problems.

Polarity analysis. Variable locks were discussed in Section 4.2.1 in the context of LP, where it was shown that the count of the number of times a variable is bound from a particular direction can be used in LP presolving. This idea can be extended to more general constraints, as done in SCIP (Achterberg, 2009; Heinz et al., 2013). The polarity of a variable is implied by the bounds of the variable, and the directions in which it is bound by the constraints of a program. The polarity can be taken into account during compilation or solving to find solutions faster. This has also been used in Brand et al. (2008), where polarity analysis was used to decide which encoding of a `maximum` constraint should be used during compilation.

k -singleton arc consistency (k-SAC). SAC, introduced in Section 2.7.2, removes values from variable domains to ensure the next decision made cannot cause a failure. A generalisation of this concept is explored in Dongen (2006), in which k-SAC is described. k-SAC reduces the domains of variables to ensure the next k decisions will lead to consistent domains. While this approach can be time consuming, it can also lead to huge reductions in the size of the search space, and its application has resulted in the closing of some previously open problems.

A related reduced form of SAC is that of *shaving*, where a variable is repeatedly assigned its bounding values, removing infeasible ones until valid bounds are found. This approach is not as thorough as SAC, but provides a good middle-ground between the cheapest propagation and the more expensive SAC.

Conclusion

CP presolving techniques such as SAC and shaving can be expensive. However, when appropriately applied they can allow problems to be solved faster. This may be the case when presolving is performed before compiling for target solvers that require large decompositions, e.g., when compiling for MIP or SAT solvers. Effective presolving for these targets requires communicating the tighter domains to the compiler so that it can make use of them.

4.2.5 Related Work in Compilation

A detailed overview of the compilation of MiniZinc models was presented in Section 2.5. Compiling a constraint model to a good MIP program can be quite difficult. Even small instances can require a large number of constraints and variables to be described correctly. These large programs can pose a difficulty for MIP solvers due to either their combinatorial complexity, or simply because they require a large amount of memory to process. Presolving performs a whole program analysis that aims to improve a program. While parts of this analysis cannot be done until the entire MIP program has been flattened, much of it can. In fact, the need for some of the low-level presolving could be avoided entirely by omitting parts of the program that are known to be redundant, from the beginning, thus improving the program and, in some cases, drastically speeding up flattening.

As we have seen, some presolving approaches focus on the reconstruction of higher level constraints that have been decomposed in a program (i.e., detecting cardinality constraints in SAT (Biere et al., 2014), and detecting permutation structures in MIP (Salvagnin, 2014)). Once detected, stronger propagation may be used to tighten domains and to remove redundant variables and constraints earlier, saving time on flattening and detection. The rest of the section reviews previous work that perform a multi-pass compilation, by first inferring information about a problem specification, and then using the inferred information to obtain a better program for the target solver.

As shown by Dittel et al. (2009), it is sometimes useful to perform presolving before compiling a model for a MIP solver. In their work, the presolving performed by a MIP solver was not sufficient to allow a problem to be solved correctly. The authors deemed that MIP presolving was not able to infer enough from their linear representation and set about implementing their own problem-specific presolving. Most notably, their approach included a technique that improved the linear representation of the problem by using hand coded algorithms for presolving the non-linear constraints which are later implemented with linear constraints. This non-linear presolving lead to a significant improvement in solving time, increasing the number of instances that could be solved. This further motivates the need for presolving higher level representations before compilation for some low-level target solver. While hand-coded algorithms were used in this case, CP solvers provide dedicated algorithms that can reason about high-level, non-linear constraints. This makes them ideal for use in presolving.

The use of CP solvers to preprocess combinatorial optimisation programs before the actual solving begins has been explored in, for example, Achterberg (2009). The *Savile Row* compiler for ESSENCE' performs multi-pass presolving using the constraint propagation engine of *Minion* (Nightingale et al., 2014) during the first pass. In addition to constraint propagation, *Savile Row* can run variants of SAC to infer stronger variable bounds. While ESSENCE' is a powerful modelling language, it does not support user-defined predicates and functions, and solver-specific translation routines are hard-coded into the compiler. Thus, the compiler can communicate presolving results through top-level model variables and those that can be uniquely identified by the expressions that introduce them (Nightingale, 2016). In contrast, the approach presented in this chapter is more general, since it is compatible with the model structure as expressed in predicate and function definitions and calls, by enabling presolving on any auxiliary variables introduced by these structures.

Another example of the use of constraint programming techniques to presolve a program before solving, was explored in Metodi et al. (2011) for improving SAT formulations. There, the equivalence of some literals in the SAT formulation of an instance was inferred by plugging a specialised constraint program into an encoding aware constraint propagation based compiler. This resulted in a large reduction in the size of encoded SAT programs and a significant improvement in solving time. This differs from the approach presented in this chapter as the compilation translates a low level constraint program to a SAT formulation, whereas our approach is for compiling high-level models to programs.

The above examples show that multi-pass presolving during compilation can be effective. The rest of this chapter introduces a new multi-pass presolving approach that makes use of model structure to increase the amount of information that can be shared between passes, leading to the compilation of more efficient programs.

4.3 Multi-Pass Presolving

This section presents the two main contributions: a technique for preserving model structure and an approach that can use this preserved structure to facilitate communication between different flattening passes. Most of the traditional presolving methods discussed in the previous sections work at the *program* level, where all variables and constraints are known. A compiler for a high-level modelling language, to get to this point, must *make decisions during flattening* that depend on its current (typically incomplete) knowledge of variable domains. For example, the compiler can only simplify a quadratic constraint $x*y$ into a linear constraint if the value of x is known at compile time; it can also turn a reified constraint $b \leftrightarrow c$ into a non-reified constraint c if b is known to be true; and it can avoid generating unnecessary code for loops such as `forall (i in lb(x)..ub(x)) (c(i))` if it can shrink the bounds of x . This case often appears with the introduction of different viewpoints, such as the introduction of an encoding of an integer variable using binary variables. These are all common situations where considerable improvements can be made when more information is made available.

One solution to this problem, is to flatten at least twice, with initial passes collecting information about the instance, and subsequent passes using this information to produce better programs. Interestingly, as seen in the previous section, the different passes do not need to flatten for the same target. The design of MiniZinc, with its solver-specific libraries of constraint definitions implemented in the MiniZinc language, rather than as opaque pieces of software, is ideal for this approach. However, multi-pass, multi-library presolving poses a significant technical challenge: the different passes may create different programs, with different variables, domains, and constraint decompositions. These programs lack the model structure needed to allow information to be correctly shared between them. Therefore, the flattener needs to identify corresponding variables across passes, and communicate information gathered about those variables from one pass to the next.

The remainder of this section presents (1) examples that demonstrate how flattening can benefit from multi-pass presolving, (2) how the multi-pass presolving can strengthen some optimisations that a compiler can use, (3) how the flattener can benefit from being able to identify and communicate more variable information across passes, and (4) how this communication of variable information can be achieved.

4.3.1 Multi-Pass Examples

The following examples show how additional information from multi-pass presolving can lead to better FlatZinc.

```

array[1..3] of var 1..30: x;
constraint all_different (i in index_set(x)) (x[i] div 3);
constraint forall (i in index_set(x)) (x[i] <= 20);

```

Listing 4.1: MiniZinc model.

```

array[1..3] of var 1..20: x;
array[1..3] of var 0..10: xi;
constraint int_div(x[1],3,xi[1]);
constraint int_div(x[2],3,xi[2]);
constraint int_div(x[3],3,xi[3]);
constraint all_different(xi);

```

Listing 4.2: FlatZinc after first pass.

```

array[1..3] of var 1..20: x;
array[1..3] of var 0..6: xi;
constraint int_div(x[1],3,xi[1]);
constraint int_div(x[2],3,xi[2]);
constraint int_div(x[3],3,xi[3]);
constraint all_different(xi);

```

Listing 4.3: FlatZinc after second pass.

Improved bounds. Listings 4.1–4.3 show a MiniZinc model and two possible FlatZinc programs. The model contains an array of variables `x` and two constraints. The first constraint restricts the result of dividing each variable in `x` by 3 to be distinct. The second one adds tighter bounds to the variables in `x`.

Listing 4.2 shows the result of flattening this model for a CP solver using single pass flattening. The first constraint introduces the intermediate variables `xi` representing the division by 3 of each variable in `x`. The bounds for the `xi` variables are computed to be in the range 0..10, as $\lfloor 30/3 \rfloor = 10$. These intermediate variables are then passed to the `all_different` constraint. After this constraint has been flattened, the flattening proceeds to the second constraint where tighter bounds for the variables in `x` are added. If this second constraint had occurred earlier, the flattener would have had tighter domains for the variables in `x` earlier and, as a result, would have been able to further tighten the bounds on the variables in `xi` as they were being introduced. This again demonstrates the difference between whole program optimisation and the typical compilation approach.

Listing 4.3 shows how the bounds could be improved by a second pass of flattening. The bounds for the `xi` variables are correctly computed as 0..6 since the new bound of 20 gives us: $\lfloor 20/3 \rfloor = 6$. Note that this is only possible if the compiler can correctly identify the equivalence between *introduced variables* in the two passes.

Shorter loops. The improvement seen in this first example may seem small, as many propagation based solvers will likely perform simple propagation at the root node that will find the tighter bounds for the `xi` variables. Assume however, that we want to solve this example using a MIP solver for which the flattener will have to encode the `all_different` constraint, using a set of zero-one variables and a set of linear constraints over these.

Such a linear decomposition of `all_different` is shown in Listing 4.4. The definition introduces an array of zero-one variables provided by the call to `eq_encode(x)`. The definition of the `eq_encode` for an array of variables also introduces a matrix of zero-one variables and, for each integer variable, calls another `eq_encode` function. This `eq_encode` introduces an array of zero-one variables and calls the `equality_encoding(x,y)` predicate to establish the connection between the integer variables and these introduced zero-one variables. It is important to note that the number of zero-one variables introduced depends

```

predicate all_different(array[int] of var int: x) =
  let {
    array of var 0..1: x_eq_d = eq_encode(x)
  } in (
    forall(d in index_set_2of2(x_eq_d)) (
      sum(i in index_set_1of2(x_eq_d)) ( x_eq_d[i,d] ) <= 1
    )
  );

predicate equality_encoding(var int: x, array[int] of var 0..1: x_eq_d) =
  x in index_set(x_eq_d)
  /\ sum(d in index_set(x_eq_d)) ( x_eq_d[d] ) = 1
  /\ sum(d in index_set(x_eq_d)) ( d * x_eq_d[d] ) = x;

function array[int] of var 0..1: eq_encode(var int: x) =
  let {
    array[lb(x) .. ub(x)] of var 0..1: y;
    constraint equality_encoding(x,y);
  } in y;

function array[int,int] of var int: eq_encode(array[int] of var int: x) =
  let {
    array[index_set(x),lb_array(x)..ub_array(x)] of var int: y =
      array2d(index_set(x),lb_array(x)..ub_array(x),
        [ let {
          array[int] of var int: xi = eq_encode(x[i])
        } in if j in index_set(xi) then xi[j] else 0 endif
        | i in index_set(x), j in lb_array(x)..ub_array(x)
        ]
      )
  } in y;

```

Listing 4.4: Encoding alldifferent with the MiniZinc linearisation library.

on the bounds of the x variable, as shown by the array indexed by $\text{lb}(x) \dots \text{ub}(x)$ (marked with a red box in Listing 4.4). Note that lb and ub are compiler builtins that return a fixed value, representing a valid, but not necessarily tight bound, during compilation. Thus, the number of variables and constraints introduced by the equality encoding depends directly on the domain sizes. Having tighter bounds on the x_i variables in the example would allow the same model to be described using fewer variables and constraints. Thus, such a two pass flattening for a MIP solver would result in a 40% smaller program.

Again, this optimisation depends crucially on the bounds of the introduced variables x_i . Only communicating bounds on top-level variables, i.e., those that occur directly in the original model, cannot lead to any improvements during the second pass, as the bounds on x have not changed.

Avoiding reification. Listings 4.5–4.9 demonstrate the benefits of stronger presolving for models containing reified constraints. The model in Listing 4.5 has three integer variables x , y , and z , with domains $\{2,4\}$, $\{2,4\}$ and $\{2,4,5\}$, respectively. The first constraint is an `alldifferent` constraint, while the second constraint introduces an implication between two expressions. Listing 4.9 shows the ideal (simplified) FlatZinc for this model when compiled for a simple constraint solver.

Listing 4.6 shows the program that results from flattening this model without any presolving. The first constraint is added directly to the FlatZinc as is. The implication is transformed to the disjunction $\neg(x+y+z=12) \vee y = \max([x, y, z])$. The negation is pushed inside the linear expression and the Boolean control variables `b0` and `b1` are introduced for both sides of the disjunction.

Listing 4.7 shows what happens when the flattener learns bounds for the top-level variable `z` by propagating the `all_different` constraint (fixing `z` to 5). The resulting FlatZinc is not different. The variable `z` has been removed from some constraints since it is fixed, but the FlatZinc still contains two reified constraints and a redundant global constraint in the form of the `max([x, y, 5])` constraint, where the variables have upper bounds lower than 5. While reified constraints allow the MiniZinc language to support much more natural descriptions of problems, they often require awkward decompositions and, in many cases, may not provide useful propagation until quite late during the search.

By performing constraint propagation on the program given in Listing 4.6 the flattener can get bounds for the introduced variables `i0`, `b0`, and `b1`. Listing 4.8 demonstrates what can be achieved by taking these new bounds into account while flattening. The assignment to `i0` allows the flattener to remove the `max` constraint. With `i0` set to 5, `b0` is trivially false leaving the disjunction with only one non-false disjunct in the root-context. Recall that being in the root-context means that the control variable `b1` must be forced to be true, allowing a non-reified version of the linear constraint to be used in place of the disjunction over the reified variable. With this additional information, the flattener will produce the desired FlatZinc shown in Listing 4.9.

Flattening this model for a MIP solver, presolving would result in a roughly 60% smaller FlatZinc program due to the complexity introduced by representing the reified constraints as linear constraints. As before, it is the information about the bounds of introduced variables that makes this kind of optimisation possible.

4.3.2 Presolving Phases

During and after a first pass of flattening, the compiler can infer different kinds of information, such as stronger variable domains and variable aliases. These can then be communicated to the presolving phases explored in the following sections. These phases seek to help the compiler produce efficient programs by simplifying constraints, cleaning up the variables and constraints of a program once flattened, and performing constraint propagation to tighten domains further.

Simplifications during flattening

Section 2.5.3 presented several optimisations that compilers for constraint modelling languages typically perform. Here we discuss the effect that multi-pass compilation can have on these optimisations.

Constant Folding: When a variable is fixed during compilation, all references to this variable should be replaced with the constant value that was assigned.

```

var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
constraint x+y+z=12 -> y=max([x,y,z]);

```

Listing 4.5: Original MiniZinc.

```

var {2,4}: x; var {2,4}: y; var {2,4,5}: z;
constraint all_different([x,y,z]);
var 2..5: i0 = max([x,y,z]);
var bool: b0 = (y = i0);
var bool: b1 = (x+y+z != 12);
constraint or(b0,b1);

```

Listing 4.6: Standard FlatZinc.

```

var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
var 2..5: i0 = max([x,y,5]);
var bool: b0 = (y = i0);
var bool: b1 = (x+y != 7);
constraint or(b0,b1);

```

Listing 4.7: FlatZinc with tightened top-level bounds.

```

var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
var {5}: i0 = max([x,y,5]); % remove (unused)
var {false}: b0 = (y = 5); % remove (entailed)
var {true}: b1 = (x+y != 7); % replace (x+y != 7)
constraint or(b0,b1); % remove (entailed)

```

Listing 4.8: FlatZinc with tightened top-level and intermediate bounds.

```

var {2,4}: x; var {2,4}: y; var {5}: z;
constraint x != y;
constraint x+y != 7;

```

Listing 4.9: Desired FlatZinc.

Variable Aliases: During compilation, the equality among several variables may be detected. If so, these equivalent variables can be replaced with a single unified variable.

Constraint Aggregation: There are several schemes by which certain constraints can be aggregated into fewer and/or simpler constraints. For example, the sets of constraints $a + b = x$, $x + c = y$, $y + d = z$ can be combined into a single constraint $a + b + c + d = z$. With more variables being fixed by multi-pass compilation, these aggregated constraints can be much more concise.

Compiling partial functions: If we learn from compilation that a constraint is total, later passes can compile the same constraint as a total function.

Common Subexpression Elimination: If two variables are fixed to the same value or found to be equal to each other, more subexpressions can also become identical and be shared.

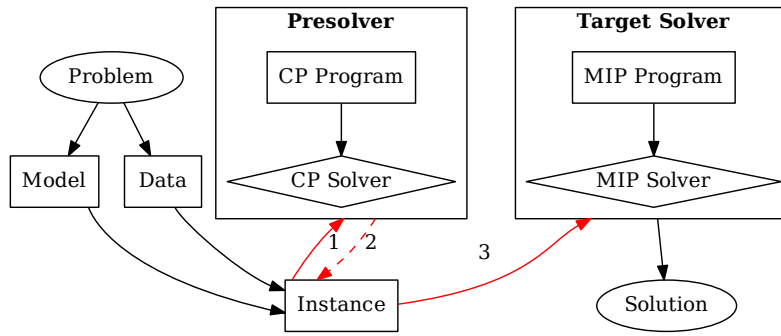


Figure 4.1: Example configuration of the multi-pass presolving approach.

Overloading Resolution: Knowledge about fixed variables allows the use of more specialised decompositions.

Linear Simplification: Fixing variables and tightening their bounds will result in shorter linear constraints.

Boolean Simplification: Fixing Boolean variables allows for simplification of Boolean expressions, fewer disjunctions, and constraints being pushed into the top-level conjunction instead of being reified.

Post-flattening code optimisation

After flattening, the compiler will again perform some basic constant propagation (replace fixed variables with fixed values and evaluate subexpressions which no longer contain variables), unify variables that have been found to be equal, and remove variables and constraints that are not used. This can help when, for example, a variable has been introduced but all constraints that referenced it have been found to be entailed, or when a reified constraint appears in a disjunction for which one disjunct has already been shown to be true.

Constraint propagation

In addition to the optimisations performed during and after flattening, the compiler can run the propagation engine of a generic constraint solver, whose specialised propagators for various global constraints can produce smaller variable domains or even fix variables. Figure 4.1 presents an example execution of this multi-pass compilation approach, configured for a two pass compilation using a CP solver as the presolver for a MIP target. Three steps are marked in this figure: The first performs an initial flattening of the instance into a CP program and gives it to the CP solver for presolving. In the second step, the information learned from the CP solver is returned and used to strengthen the instance. Finally, in the third step, this strengthened instance is flattened into a MIP program and given to the MIP solver for solving.

4.3.3 Preserving Structure

In the examples of Section 4.3.1, the set of variables generated in the first pass is the same as the set generated in the second pass. If this were guaranteed, the compiler could update the domains of variables by simply matching the variable names. Unfortunately, this is typically not the case in practice. Indeed, while bounds can be easily communicated for top-level variables that are common between different FlatZinc programs, the compiler cannot rely on the temporary variables introduced during flattening having the same names across different compilations. The code in Figure 4.2 illustrates the problem of identifying variables across different compilation passes.

The first MiniZinc file, `a.mzn`, defines a predicate $f(k)$ that introduces an integer variable x and a constraint $h(x, k)$, which is defined as a builtin constraint. The second file, `b.mzn`, includes `a.mzn` and defines a predicate $g(j)$ that calls $f(k)$ and $f(k+1)$, as well as two constraints that call $g(4)$ and $f(1), f(2), f(3)$, respectively.

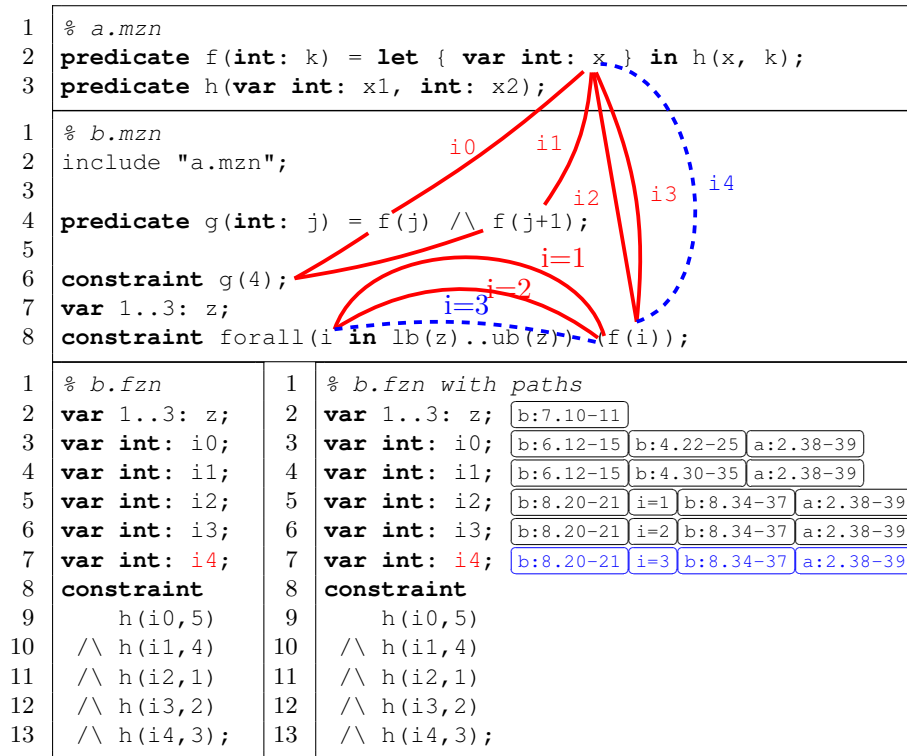


Figure 4.2: Preserving explicit structure to the program-level.

The flattening of model `b.mzn` will introduce five different instances of variable x . The MiniZinc compiler will simply number them in the order they are generated. The first two, $i0$ and $i1$, come from the call to $g(4)$ in line 6, which results in calls to $f(4)$ and $f(5)$, and which in turn result in the calls $h(i0, 5)$ and $h(i1, 4)$. The remaining three are generated on line 8, where the bounds of z that are known to the compiler are iterated over. The resulting FlatZinc file, `b.fzn`, appears at the bottom left of Figure 4.2.

Let us now have a look at the variable names generated in different flattening passes. If the second pass produced exactly the same FlatZinc code as the first pass, remembering the *name* of each variable, such as $i2$ in our example, would be sufficient. But assume

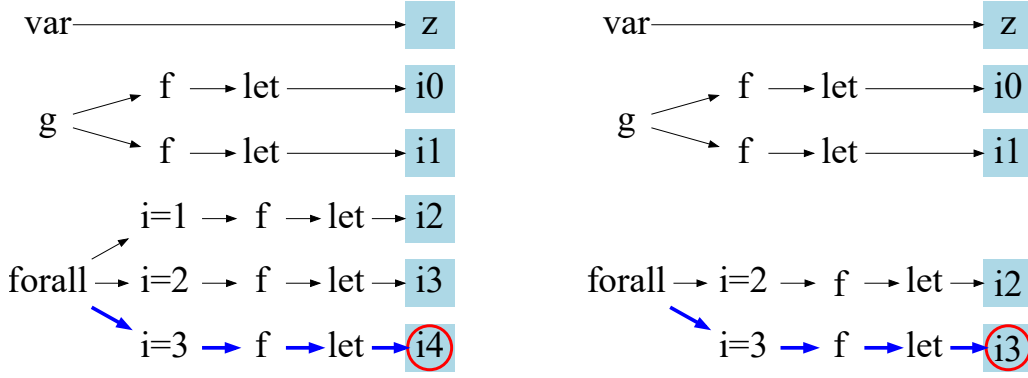


Figure 4.3: Trace of compilation for introduced variables showing how paths can match variables when simple identifiers cannot.

that, after the first pass, presolving managed to narrow the domain of z to the set $2..3$. In the second pass, iteration now starts at $i=2$, and only four instances of x are generated (i_0 through i_3). Crucially, the names of the variables will change, too: the name i_2 will now refer to the variable from iteration $i=2$, whereas in the first pass it referred to iteration $i=1$. Clearly, it would be wrong to transfer any information inferred about i_2 during first pass presolving to the new second-pass variable i_2 . Instead, information about first-pass i_3 now corresponds to second-pass i_2 .

4.3.4 Variable Paths

The above example illustrates how a multi-pass, multi-target presolving algorithm needs to *identify* variables that arise from the same subexpressions *across passes*. The algorithmic solution to this problem is to generate variable identifiers that are unique and that do not depend on the order of flattening or on what has been flattened before. To satisfy these conditions, it is sufficient for identifiers to capture the following information:

- The **call stack** leading to the introduction of a variable. For example, variable i_1 in Figure 4.2 was added by the call to g on line 6, which then called $f(j+1)$ on line 4.
- Any **loop iteration values**. For example, variable i_3 was introduced when i had value 2.

We call an identifier that captures this information a **variable path**. The variable paths for the example in Figure 4.2 are represented graphically, with lines connecting each predicate call through to the place where the variable is introduced. The matching made possible by this approach is demonstrated in Figure 4.3, where the two separate compilations of this model are presented. First, with the original bounds on z being in the range $1..3$, and then $2..3$. The old i_2 variable is nowhere to be found in the second compilation, and that the old i_4 corresponds to the new i_3 by matching the paths (shown here with thick blue lines).

Path representation. A textual representation can use the location in the source code to represent call sites and variable declarations, together with the value of loop variables to identify the iteration. A path for i_1 would include the location of the call $g(4)$ in file `b.mzn` on line 6, column 12–15, the location of the call $f(j+1)$ on line 4, column 31–36, and the location of x in `a.mzn`, line 2, column 38–39, resulting in the path `b.mzn:6.12-15``b.mzn:4.31-36``a.mzn:2.38-39`. For i_3 we need to add the loop iteration value: `b.mzn:8.20-21``i=2``b.mzn:8.34-37``a.mzn:2.38-39`. The bottom right code of Figure 4.2 shows the FlatZinc variables annotated with their respective variable paths. Note how the identifiers are independent of the order in which the constraints are flattened, and of the concrete bounds of the variables at the time of flattening. For instance, variable i_3 is linked to iteration $i=2$, independently of whether the loop started at 1 or 2. An approach for implementing this scheme is presented in Section 4.4.2.

4.4 Implementation in MiniZinc

This section discusses the implementation and the challenges of integrating path-based multi-pass presolving into the MiniZinc compiler presented in Chapter 2.

4.4.1 Compilation in MiniZinc

Recall from Section 2.5 that MiniZinc 2.0 is based on `libminizinc`, a C++ library that implements parsing, type-checking and flattening of MiniZinc to FlatZinc. The parser turns a text model into a set of expressions, which are then passed to the type-checking and flattening algorithms. Flattening is implemented as a recursive call-by-value (inside-out) translation, and some expressions in a model may not always be traversed during flattening, while others may be visited several times. For example, given an if expression, the flattener may traverse the `then` branch or the `else` branch, depending on the result of evaluating the condition, and an expression may be traversed several times in the case of a list or set comprehension. Visiting the same expression several times means that several items may be added to the FlatZinc program from the same syntactic position of the expression. This needs to be taken into account when computing variable paths.

4.4.2 Implementing Variable Paths

When generating large programs, e.g., when compiling for MIP solvers, it is not uncommon to introduce tens of thousands of variables. Therefore, an actual implementation must make both the generation and lookup of paths efficient. To achieve this, instead of encoding the path into the variable identifier, the implementation keeps a separate *path map* from paths (represented as strings) to variables. The initial compilation passes populate the map. If a variable to be introduced exists in the map, the pass can access all the information collected during previous passes about this variable.

The *call stack* is represented explicitly inside the flattener. This is achieved by pushing each call being processed onto the call stack. The original purpose of this call stack was to help the compiler produce useful error messages. Consider Listing 4.10, which shows

```

MiniZinc: evaluation error:
  mknapsack_global.mzn:44:
    in call 'forall'
    in array comprehension expression
      with i = 1
  mknapsack_global.mzn:45:
    in call 'knapsack'
    /usr/share/minizinc/std/knapsack.mzn:17:
    in call 'assert'
  Assertion failed: index set of weights must be equal to index set of
  profits and index set of items

```

Listing 4.10: Error in faulty model.

an error message produced by the MiniZinc compiler when compiling a model. A trace of the compilation is displayed to identify where an error occurred. It begins on line 44 of the `mknapsack_global.mzn` model. The compiler was processing iteration `i = 1` of the `forall` loop. Inside this, a call to the `knapsack` constraint is being processed. The failure occurs somewhere within the decomposition of the `knapsack` constraint in `knapsack.mzn` in the standard library, where an `assert` has failed. Finally, the error is displayed.

In order to generate error messages like these, the call stack includes information about the recursive structure of the expression that is being compiled, including all comprehensions that introduce loop variables (`with i = 1`). This data is reused by our implementation to construct variable paths.

Integrating multi-pass presolving with variable paths into the MiniZinc compiler presented in Section 2.5, can be achieved by extending the implementation of `newVar` (Algorithm 2.4) to manage a map of variables that have already been seen and their paths, as shown in Algorithm 4.1. This algorithm starts by calling the `getCurrentPath` procedure from Algorithm 4.2, which steps through the callstack and, for each callstack item c , appends to the path a pair $\langle \text{location}(c), \text{desc}(c) \rangle$ with the location and a simple descriptor for c . If the returned path is not in the *PathMap* yet, a new variable is introduced to the FlatZinc and is added to the *PathMap*, along with an integer signifying which pass this variable comes from. If a variable already exists with this *path* in the *PathMap*, the procedure checks whether the pass in which this variable was introduced was the current pass. If that is the case, it simply returns the variable from the *PathMap* without introducing any new variable. If however the variable was added in a previous pass, a new variable is constructed for this pass and the domain information from the old variable is transferred (by intersection with its current bounds) to this new variable, initialising it with the tightest bounds available.

During compilation one variable may be found to be equivalent to another. If so, the compiler will *unify* these variables into a single program level variable. This exposes a minor inefficiency in the multi-pass approach as described above. The paths for a unified variable can potentially point to a variable that is never added to the program in the first pass. Therefore, we will only see the original bounds for this variable when it is being introduced in a later pass and, thus, we must wait for it to be unified. The multi-pass

Algorithm 4.1 Integrating variable paths into compilation with unification highlighted.

Compiler State:*PathMap*: map: *path* \rightarrow $\langle \text{var}, \text{pass} \rangle$ \triangleright Mapping of paths to variables and pass number*CurrentPass*: integer \triangleright Current pass number*ReversePathMap*: map: *expression* \rightarrow *path* \triangleright Allows duplicate paths**procedure** newVar(*v*) \triangleright *v* is the name of the new variable*path* \leftarrow getCurrentPath()**if** *path* $\neq []$ **then** $\langle v', p \rangle \leftarrow \text{PathMap}[\text{path}]$ **if** $v' \neq \perp$ **then** \triangleright We have an entry for this path**if** $p = \text{CurrentPass}$ **then** \triangleright Unify if it's from the same pass $\bar{v} \leftarrow v'$ **else** \triangleright Variable from previous pass $\bar{v} \leftarrow \text{new } v$ \triangleright Create new variable $\text{dom}(\bar{v}) \leftarrow \text{dom}(\bar{v}) \cup \text{dom}(v')$ \triangleright Update domain $\text{PathMap}[\text{path}] \leftarrow \langle \bar{v}, \text{CurrentPass} \rangle$ \triangleright Add new variable to map**if** $\text{name}(v') \neq \text{canonicalName}(v')$ **then** \triangleright Was this unified previously? $\text{path}' \leftarrow \text{ReversePathMap}[\text{canonicalName}(v')]$ \triangleright Get the real path $\text{PathMap}[\text{path}'] \leftarrow \langle \bar{v}, \text{CurrentPass} \rangle$ \triangleright Point both paths at \bar{v} $\text{ReversePathMap}[\bar{v}] \leftarrow \text{path}$ \triangleright Update *ReversePathMap***else** \triangleright We have not seen this variable before $\bar{v} \leftarrow \text{new } v$ $\text{PathMap}[\text{path}] \leftarrow \langle \bar{v}, \text{CurrentPass} \rangle$ \triangleright Create new variable and $\text{ReversePathMap}[\bar{v}] \leftarrow \text{path}$ **return** \bar{v} **return new** *v* \triangleright Return new variable

approach should be able to unify variables in the *PathMap*, so that paths for unified variables will point to a single canonical variable.

Algorithm 4.1 highlights with a red box the specific checks related to unification. An extra map, the *ReversePathMap*, is required for this. It maps an arbitrary expression (just variables in this chapter) to its path in the *PathMap*. This allows several expressions to share a path and, thus, they all point to the same variable in the *PathMap*. When MiniZinc unifies a variable it creates a link between this variable's unique identifier and the canonical variable, to ensure any use of the identifier of any unified variables points to a single variable. To simulate this behaviour in the algorithm we introduce two procedures, $\text{name}(v)$ and $\text{canonicalName}(v)$, which return the identifier for *v* and the identifier for the variable to which *v* is actually bound. When a variable is retrieved from the *PathMap* its name compared to its canonical name. If they are different, we then configure the *PathMap* so that the paths for either this variable or its unified version will point to the same canonical variable from now on. This ensures that, when a previously unified variable is being added, we can just return its canonical variable without introducing a new one.

Path representation. The current implementation encodes paths into strings. This allows the *PathMap* to be implemented by a simple hash map. While more incremental

Algorithm 4.2 Procedure that returns a *path* uniquely identifying this compilation unit.

```

procedure getCurrentPath()
  path  $\leftarrow []$ 
  for each ( $i \in |CallStack| \dots 1$ ) do                                 $\triangleright$  Read CallStack backwards
     $c \leftarrow CallStack_i$ 
    path  $\leftarrow path + \langle location(c), desc(c) \rangle$ 
  return path

```

and compact data structures, such as prefix trees, could be used for this purpose, current experiments with the optimisations presented in the rest of this section suggest that the performance of the string-based solution is sufficient.

Optimised path generation. Since constructing and comparing paths as strings can be costly, the implementation employs two simple optimisations to reduce the number of paths generated. Firstly, it remembers the maximum call stack depth from previous passes. If it is in its final pass and detects a variable being introduced deeper in the call stack than the recorded maximum, it knows the variable cannot be present in the map. This will often occur when using a CP solver for the presolving pass and a MIP solver as the target. The amount of decomposition required to flatten for a CP solver will often be much less than that required for a MIP solver. For example, compiling an `alldifferent` constraint for a CP solver is as simple as just posting the constraint, whereas for a MIP solver the constraint must be decomposed, with many variables and constraints being introduced. In this case, no paths will be constructed for the introduced variables, as the compiler knows that (a) it has not seen variables at that depth before, and (b) there will not be any further passes that might need to compare paths with these. If, instead, it is not in its final pass, paths will be collected, as they may be needed in these later passes. The same can be said for variables introduced from a file that was not used in earlier passes, since no presolving information can exist about them. Again, this occurs when different passes use different libraries.

In both cases, we save the overhead of constructing the path and looking it up in the map. These optimisations are critical for large models. An optimised `getCurrentPath` procedure is presented in Algorithm 4.3, which adds *MaxPathDepth* and *VisitedFiles* to the flattener state and implements the above optimisations.

4.5 Experimental Evaluation

The experiments summarised in this section measure the impact of the new approach on compile-time overhead, program size and solving performance, when compiling for a MIP target solver and for a CP target solver. The experiments use the implementation discussed in Section 4.4 configured to perform a two pass compilation. In addition to the simplifications performed by the MiniZinc compiler (as discussed in Section 4.3.2), the implementation invokes the propagation engine of the Gecode (Gecode Team, 2006) constraint solver (version 4.3.1) in the first pass, storing tighter domains for use in the

Algorithm 4.3 `getCurrentPath` with optimisations.**Flattener State:***CallStack*: stack*PathMap*: map: *path* \rightarrow $\langle var, pass \rangle$ *CurrentPass*: integer*NPasses*: integer \triangleright Number of passes to be completed*MaxPathDepth*: integer \triangleright Length of deepest path*VisitedFiles*: set \triangleright Files visited in previous passes**procedure** `getCurrentPath()`*path* $\leftarrow []$ **if** $|CallStack| > MaxPathDepth$ **then** \triangleright If we are deeper than any previous path**if** *CurrentPass* = *NPasses* **then return** [] \triangleright If this is the final pass*MaxPathDepth* $\leftarrow |CallStack|$ **for each** ($i \in |CallStack| \dots 1$) **do***c* $\leftarrow CallStack_i$ **if** `filename(c)` $\notin VisitedFiles$ **then** \triangleright If we have not seen this file before**if** *CurrentPass* = *NPasses* **then return** [] \triangleright If this is the final pass*VisitedFiles* $\leftarrow VisitedFiles \cup \{\text{filename}(c)\}$ *path* $\leftarrow path + \langle \text{location}(c), \text{desc}(c) \rangle$ **return** *path*

second pass. Many constraint solvers utilise a trailing-based search in which the actions of a propagator must be recorded so that the propagation can be undone during backtracking. Gecode, on the other hand, primarily relies on a copying-based approach. As a result, computing a single fixed-point without doing any search does not incur this extra overhead. This makes it ideal for presolving, since no search will be performed. The first compilation pass uses the MiniZinc library for Gecode, which supports many of the global constraints provided by MiniZinc natively. The second pass uses the target solver's MiniZinc library.

For these experiments, 300 instances were taken from three years (2012, 2013, 2014) of the MiniZinc Challenge (Stuckey et al., 2010). This includes 49 different MiniZinc models which cover 46 different problems (listed in Figure 4.4). The 300 instances are those used in the challenge in those years. See Appendix A for more details. Each instance was compiled using the MiniZinc 2.0 compiler both with a single pass compilation (non-presolved) and a two pass compilation (presolved). The experiments were executed on a cluster, where each node had dual 2.00GHz Intel Quad Core Xeon E5405 processors with 16GB of memory. Compilation and solving were, however, both limited to 8Gb of memory. Solving was also limited to a maximum of 1800 seconds. The following two subsections explore the evaluation of the approach when targeting MIP and CP solvers, respectively. In addition, the last subsection presents a case study to demonstrate how the multi-pass approach can be used in practice.

4.5.1 Two Pass Compilation: For MIP

Table 4.1 shows a summary of the results of using the implemented two pass compilation for the MIP solver CPLEX version 12.6. The table shows the median (Med%) and the

Amaze-2	Amaze-3	Black-hole	Cargo-coarse
Carpet-Cutting	Celar	Fastfood	Fillomino
Filter	FJSP	FM	Ghoulomb
Handball	Jp-encoding	League	Linear2program
Mario	Mknapsack	Mqueens	MSPSP
NMSeq	Nonograms	OC-Roster	Openshop
Parity-learning	Patternset-mining	Pentominoes-int	ProjectPlanner
Radiation	RCMSP	RCPSP	Rect-packing
Road-Cons	Rubik	SB	Ship-Schedule
Smelt	Spot5	Still-life	TP
TPP	TPPPV	Train	Trip
VRP	WCSP		

Figure 4.4: Problems selected for use in evaluation of multi-pass approach.

	<i>N</i>	<i>Var</i>	<i>Con</i>	<i>Dom</i>	<i>Cmp</i>	<i>Pre</i>	<i>Sol</i>	<i>Tot</i>
Med%	250	100	100	98	102	98	-	-
	130	97	97	91	96	90	100	100
	67	97	96	86	91	87	74	79
Geo%	250	96	97	89	106	86	-	-
	130	92	93	88	97	85	72	78
	67	88	90	85	96	80	53	61

Table 4.1: Two pass compilation for a MIP solver experiment.

geometric mean (Geo%) of the percentage sizes and runtimes of presolved versus non-presolved programs. Three subsets of the 300 instances are reported: the set of instances that compile without exceeding memory or time limits, showing the impact on compilation time; the set where presolving has changed the number of variables or constraints after MIP presolving, showing the potential impact on program size; and the set where solve times for both presolved and non-presolved programs are greater than 1 second and less than the 1800 second time limit. This organisation is based on the difficulty in reasoning about results when either solve times exceed the limit, or are so small that differences can be due to environmental factors (e.g., process scheduling).

Column *N* shows the number of instances in each group. Columns *Var*, *Con*, and *Dom* show the percent size of presolved programs relative to non-presolved ones, measured in number of variables, constraints, and the product of domain sizes, respectively. The columns *Cmp*, *Sol*, and *Tot* denote the percentage duration of compilation, solving and the total compile and solve time of presolved programs relative, again, to the non-presolved ones, respectively. Solve times are only available for instances where the number of constraints and variables were changed by presolving. In addition the *Pre* column shows how long it took CPLEX's own built-in presolver to prepare the program before solving began. Figure 4.5 shows more detail regarding the impact on each of the 130 instances. In particular, it shows the average total (compile + solve) time for presolved and non-presolved programs, along with the ratio between them. Blue arrows show how the average total solve time for each instance was changed by the two pass compilation. Arrows that point down (up) are faster (slower) with presolving. Times are marked on the right-hand axis.

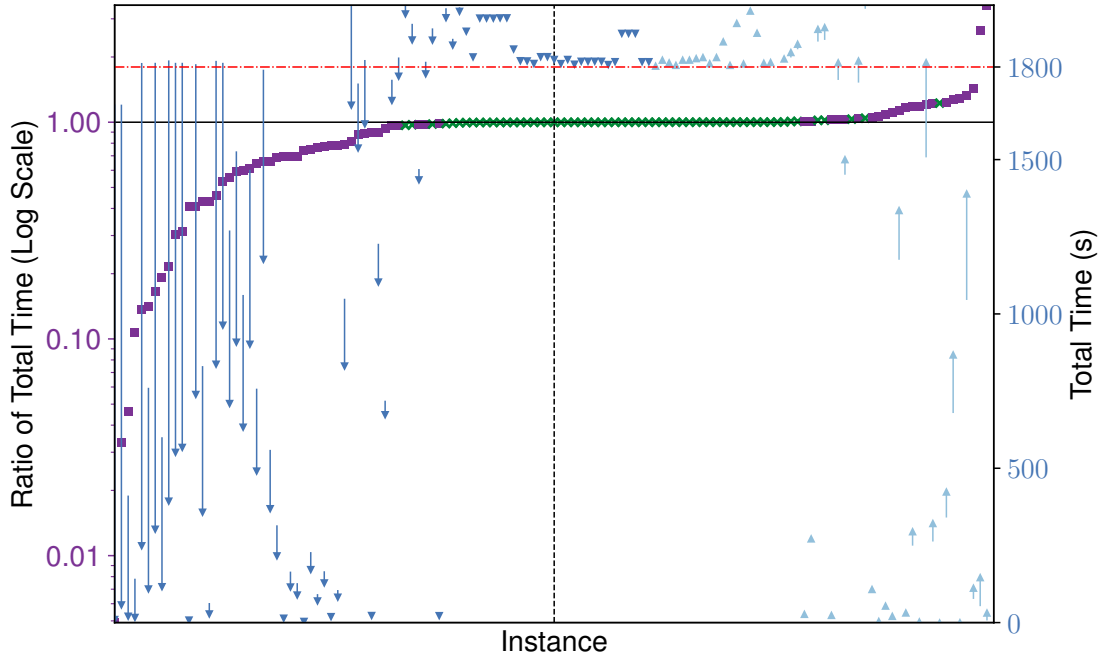


Figure 4.5: Comparison of single and two pass compilation for a MIP solver.

Instances are sorted in order of ratio, with the smallest ratio (biggest improvement) first for clarity. The ratios are represented by purple squares and green X s and are presented on a log scale to emphasise the impact of presolving. Squares are from the set of 67 comparable instances, and X s are either too fast or too slow to be compared meaningfully. A red dot-dashed line indicates the 1800 second solving timeout. Note that the total time includes compilation and so we can see several points that occur above this line. The horizontal line at the 1.0 ratio indicates where presolving has had little to no impact. Additionally, a dashed vertical line marks the median total time.

To control for the erratic behaviour of MIP solvers (Fischetti et al., 2014), each instance is solved six times with different random seeds. To narrow the set of 250 compilable instances, we looked at the results of running CPLEX’s presolver on each program, and kept the 130 instances where multi-pass presolving resulted in a change to the number of variables or constraints *after* MIP presolving. This is because in those 120 cases MIP presolving reached the same point regardless of whether we presolved beforehand or not.

As expected, we see that the two pass compilation has a big impact when applied to MIP. The total time improves by 39% (geometric mean) in the 67 comparable instances. This improvement is due to the solve time being almost halved on average. Thanks to the relatively low cost of compiling for CP, presolving has only a small cost of around 6% for the whole set of instances. Interestingly, it even speeds up compilation in many cases, most likely due to a reduction in the amount of decomposition required.

Figure 4.5 shows these results more clearly. Most of the instances with close to 1.0 ratios are there because solving timed out in both the case of single pass and two pass

Model	N	Var	Con	Dom	Cmp	Pre	Sol	Tot
Amaze-2	1	91	92	85	88	87	129	128
Amaze-3	8	94	94	88	89	92	108	102
Black-hole	2	89	99	98	100	195	111	111
Fastfood	1	99	99	59	60	3	79	77
Fillomino	4	85	86	77	83	79	82	81
FJSP	1	100	100	96	119	133	95	107
Jp-encoding	5	64	76	75	86	65	23	72
Linear-to-program	1	98	96	92	106	91	79	79
Mario	7	98	97	85	90	75	48	49
OC-Roster	2	101	101	92	97	84	96	87
Pattern-set-mining-k1	2	92	92	102	160	89	12	39
Pattern-set-mining-k2	2	24	25	33	121	27	56	91
Project-Planner	3	99	99	97	122	97	119	119
Cyclic-RCPSP	4	81	89	68	82	70	116	111
Rect-Packing	1	100	100	89	90	91	178	143
Road-Construction	3	104	102	176	147	115	6	8
SB	9	99	100	85	100	98	65	66
Still-life	2	100	100	98	110	96	97	97
TPP	7	97	96	83	86	72	14	17
Train	2	88	90	81	82	98	84	82

Table 4.2: Per-model summaries for MIP experiment.

compilation. Note that the cost of compilation is minimal in these cases, as it is only a tiny fraction of the total time. The number of points that lie below the black line is quite large, with the total time reduced to less than half in many cases and three instances where there is an order of magnitude improvement. We can see from the blue arrows that many of these improvements are indeed quite large, with several instances that time out without presolving, being solved in less than 500 seconds. Above the line we see a handful of instances that take longer, with only two having 1.5 times (150%) slower total time. The upward-pointing blue arrows show that, in these cases, the slow-down is typically not as significant, with only a small handful of cases where two pass compilation led to a timeout.

Table 4.2 presents a model-by-model breakdown of the results for the set of 67 instances. Here the N indicates how many instances are in the set for each model. Some of the interesting cases are discussed below:

FastFood: Presolving has little impact on the number of variables and constraints in the resulting programs, removing just 1% of them. Domains however, are reduced significantly. The impact of this is a much faster compilation and a hugely sped up MIP presolve (3%), indicating that the MIP solver had to work hard to remove redundant information in the non-presolved case.

Pattern-set-mining-k1 and k2: The k in the model names represent different versions of the model, which mine sets containing k items. With $k = 1$, solve times are seen to be significantly improved, despite only a small reduction in the number of variables and constraints. However, in the $k = 2$ case, program size can be greatly reduced

resulting in programs that are 25% the size of non-presolved programs. This does translate into much faster solving (44% faster), but presolving is so expensive that the overall improvement is relatively small (9% faster total time).

Road-Construction: This model was one of the most expensive to presolve, taking 147% of the duration of the non-presolved case. Interestingly, the number of variables and constraints actually increased due to two pass compilation. This is likely due to extra holes being introduced in the domains of some variables, requiring more constraints to represent these in the MIP formulation. The domain size increases significantly due to these added binary variables. These instances are shown to benefit the most from presolving. This is due to some numerical instability that the MIP solver encounters when faced with the non-presolved program. In fact, for this model, CPLEX works much harder to find solutions, and returns invalid solutions when two pass presolving has not been applied. Activating CPLEX’s “numerical precision emphasis” parameter allows these instances to be solved correctly indicating that it is not the fault of the compiler but the default configuration of the solver.

Rect-Packing: This is the model that seems to be impacted least favourably by two pass compilation. Solving takes 78% longer when presolved. The exact reasons for this have been difficult to discern. Changing CPLEX’s parameters again reveals different results. For example, enabling the “numerical precision emphasis” in CPLEX makes the presolved instances faster than non-presolved.

Examination of the results suggest that the biggest improvements are gained when presolving can reduce the number of `element` and reified linear constraints in an instance. Overall, the results indicate that this presolving technique is effective when compiling for a MIP target, and suggest that multi-pass compilation should be the default compilation approach whenever a user is targeting a MIP solver.

4.5.2 Two Pass Compilation: For CP

Table 4.3 shows a summary of the results when using the implemented two pass presolving for the CP target solver Chuffed (Chu, 2011), a lazy clause generation solver that has often performed well in the MiniZinc challenge. This solver was selected, instead of Gecode, because it supports a different subset of global constraints and, thus, its propagation could be complementary to Gecode’s. As mentioned above, Gecode performs a copying-based

	<i>N</i>	<i>Var</i>	<i>Con</i>	<i>Dom</i>	<i>Cmp</i>	<i>Sol</i>	<i>Tot</i>
Med%	271	98	100	96	188	-	-
	132	90	92	91	194	100	101
	57	89	95	89	195	98	103
Geo%	271	90	92	85	170	-	-
	132	84	88	76	179	97	112
	57	85	89	75	187	93	105

Table 4.3: Two pass compilation for a CP solver experiment.

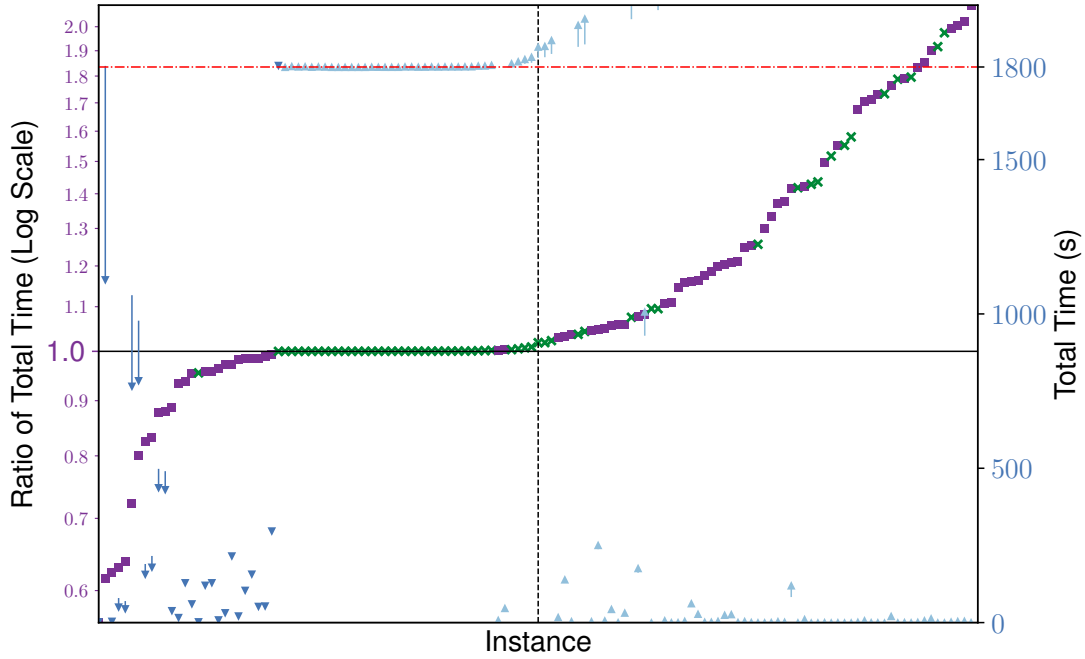


Figure 4.6: Comparison of single pass and two pass compilation for a CP solver.

search. When copying, the solver simplifies the program somewhat, removing constraints that will no longer propagate and replacing some constraints with more appropriate decompositions where appropriate. Thus, Gecode performs much of the same work that a second compilation pass would perform making it unnecessary.

Table 4.3 is organised in the same way as Table 4.1, except for the *Pre* column, which is not present here since Chuffed does not have a separate presolve step. Similarly, the accompanying plot in Figure 4.6 is organised in the same way as that of Figure 4.5.

For Chuffed, some reduction in program size and solving time was expected, as it supports fewer global constraints than Gecode and, thus, should benefit from the stronger propagation. A significant increase in relative compilation time was also expected, since two similar compilations must be performed.

Table 4.3 shows that, on the full set of 271 instances, we often have a reduction in problem size with a typical compilation time increase of 70%. The compile time increases with the impact of presolving, with an increase of 87% in the group of comparable instances. Solve time is often reduced but often not enough to compensate for the compilation time.

Figure 4.6 shows a different picture from the MIP experiments. Here we see far more instances above the line, meaning that the total time is longer with two pass compilation. However, looking at the upward pointing arrows we see that many of these cases are really only a few seconds slower, indicating that the slow down is due to the cost of compiling twice and not from solving being much slower. However, some of them are due to the programs being harder to solve. The presolving does allow several instances to be compiled and solved faster but, in general, it has little impact. Three down-pointing

Model	N	Var	Con	Dom	Cmp	Sol	Tot
Amaze-2	1	89	94	94	119	201	190
Amaze-3	4	95	95	96	146	119	126
Black-hole	2	99	100	93	127	100	100
Fastfood	4	95	96	81	170	88	88
Fillomino	3	88	92	90	197	68	68
Ghoulomb	3	85	84	84	166	97	102
Handball	5	84	87	81	211	57	80
Mario	6	73	70	64	189	99	99
Nonograms	3	93	100	93	196	99	119
OC-Roster	1	94	97	89	182	114	116
Pattern-set-mining-k1	1	98	100	98	199	98	105
Pattern-set-mining-k2	1	45	53	49	186	98	177
Radiation	2	51	59	16	170	89	106
Cyclic-RCPSP	3	86	87	66	187	91	94
Ship-schedule	6	100	100	100	223	99	135
Still-life	5	97	98	97	214	104	117
TPP	2	93	91	91	195	102	120
Train	1	80	88	94	180	98	99
VRP	1	65	98	71	196	83	83
WCSP	3	74	100	27	200	91	96

Table 4.4: Per-model summaries for CP experiment.

arrows stand out as having improved significantly. In little more than half of the cases, presolving actually hurts performance, even doubling the total time in one case.

Further analysis of the results identified two classes of problem where presolving appears to hurt performance. The first case is when the total time is dominated by compilation time. In this case, compiling twice adds a significant overhead. We can address this issue by limiting the time available for presolving. The second case occurs when solving the presolved model takes much longer than the non-presolved. Here the presolving appears to have some negative effect on the search strategy and, possibly, on the nogoods learned during search. It would be useful to further investigate how to detect these cases. However, this investigation is made difficult by the sensitivity of different solvers to slight changes in input and parameters (Fischetti et al., 2014).

Table 4.4 presents a model-by-model breakdown of the results. Some of the interesting cases are discussed here.

Amaze-2: Despite reducing the size of the program to be solved, presolving appears to make solving much harder. In exploring this behaviour by hand, it was discovered that modifying the search strategy can change whether two pass compilation is beneficial or not. The model defines a search strategy that labels two specific arrays of variables before any others. Placing one array first makes presolving hurt performance, while placing it second makes presolving increase performance. Enabling random tie-breaking in Chuffed, removes this difference in performance.

Fillomino: Presolving successfully reduces the size of these programs by about 10%. Achieving this reduction costs twice the compilation time. Despite this, the solve time was improved enough for the total solve time to be improved by 32%.

Handball: Presolving again doubles the cost of compilation time and, again, improves solving enough for the total solve time to be improved by 20%.

Pattern-set-mining-k1 and k2: For $k=1$ we see little impact of the two pass compilation. However, for $k=2$ we see that the constraint programs are about half the size when compiled using the two pass compilation. However, this does not translate into faster total time. Solve time was only improved slightly and the cost of compilation was high in comparison, leading to a large overhead in total time.

VRP: Despite compile time being doubled, for this model we see a large reduction in the number of variables, with solve time also being reduced. However, the compile time was small in comparison to the solve time and, thus, had little effect on the total time.

WCSP: Another case where compilation time was doubled. However, solving time was reduced by 9%, which was enough to pay for the compilation time.

In these experiments we see that using the multi-pass compilation approach when targeting a CP solver (or maybe just a solver that is similar to the presolving solver) is not often worth it. This presents a counter argument to the suggestion that multi-pass compilation should be a default setting. Additionally, from the case of the **Amaze-2** instance, if multi-pass compilation was the default, a user may be forced to reason about how this may help or hinder their chosen search strategy, potentially adding unnecessary cognitive overhead to modelling.

4.5.3 Two Pass Compilation: Case Study

To demonstrate how path-based multi-pass presolving can be used in practice, a case study is presented here for a model that produces large MIP formulations. The Resource Constrained Modulo Scheduling Problem (RCMSP) is a cyclic resource constrained project scheduling problem with generalised precedence relations, scarce cumulative resources, and tasks that are repeated infinitely. The objective is to find a cyclic schedule that first minimizes the period of the schedule and then the makespan.

The model used (shown in Listing 4.11) is mainly comprised of linear constraints, reified logical expressions and a set of `cumulative` constraints. The objective is also defined in terms of non-linear `min` and `max` constraints over a set of variable expressions. Compilations of this model for MIP result in long FlatZinc programs, mainly due to the large overhead introduced by the decomposition of the `cumulative` constraint (described in Section 2.3.4). To give an idea of how large this compilation can be, compiling this constraint, for just 10 tasks, with start times, durations, and requirements in the range 1..10, results in 1,211 variables and 1,420 constraints.

```

1  % rcmsp.mzn
2  include "cumulative.mzn";
3  int: n_res; set of int: Res = 1..n_res; array [Res] of int: rcap;
4  int: n_tasks; set of int: Tasks = 1..n_tasks;
5  array [Tasks] of int: d =
6    [ if i in {1, n_tasks} then 0 else 1 endif | i in Tasks ];
7  array [Tasks, Res] of int: rreq;
8  int: n_prec; set of int: Prec = 1..n_prec;
9  array [Prec, 1..4] of int: prec;
10 int: t_max = sum(i in Prec where prec[i, 3] > 0)(prec[i, 3]);
11 set of int: Times = 0..t_max; set of int: ITERS = 0..n_tasks;
12
13 array [Tasks] of var Times: s; array [Tasks] of var ITERS: k;
14 var Times: makespan =
15   max([ s[i] - k[i] * s[n_tasks] | i in Tasks where i > 0 /\ i < n_tasks ])
16   - min([ s[i] - k[i] * s[n_tasks] | i in Tasks where i > 0 /\ i < n_tasks ])
17   + 1;
18 var 0.. (t_max * (1 + t_max)): objective = s[n_tasks] * t_max + makespan;
19
20 constraint forall(i in Prec)(
21   let { var bool: b } in (
22     ( b <-> ( s[prec[i,1]] + prec[i,3] <= s[prec[i,2]] ) )
23     /\ k[prec[i,1]] + not(b) <= k[prec[i,2]] + prec[i,4]
24   ));
25
26 constraint forall(i, j in Tasks where i < j)(
27   if exists(r in Res)(rreq[i, r] + rreq[j, r] > rcap[r])
28   then s[i] + d[i] <= s[j] /\ s[j] + d[j] <= s[i]
29   else true
30   endif);
31
32 constraint forall(r in Res)(
33   let {
34     set of int: ResTasks =
35       { i | i in Tasks where rreq[i, r] > 0 /\ d[i] > 0 },
36     int: sum_rreq = sum([0] ++ [rreq[i, r] | i in ResTasks])
37   } in (
38     if sum_rreq <= rcap[r]
39     then true
40     else cumulative(
41       [ s[i] | i in ResTasks ],
42       [ d[i] | i in ResTasks ],
43       [ rreq[i, r] | i in ResTasks ],
44       rcap[r])
45     endif));
46
47 constraint k[n_tasks] = max([ k[i] | i in Tasks where i < n_tasks])
48 /\ forall(i in Tasks where i < n_tasks)(
49   s[i] + d[i] <= s[n_tasks]);
50
51 constraint s[1] = 0 /\ k[1] = 0;
52
53 solve minimize objective;

```

Listing 4.11: Model for RCMSP problem.

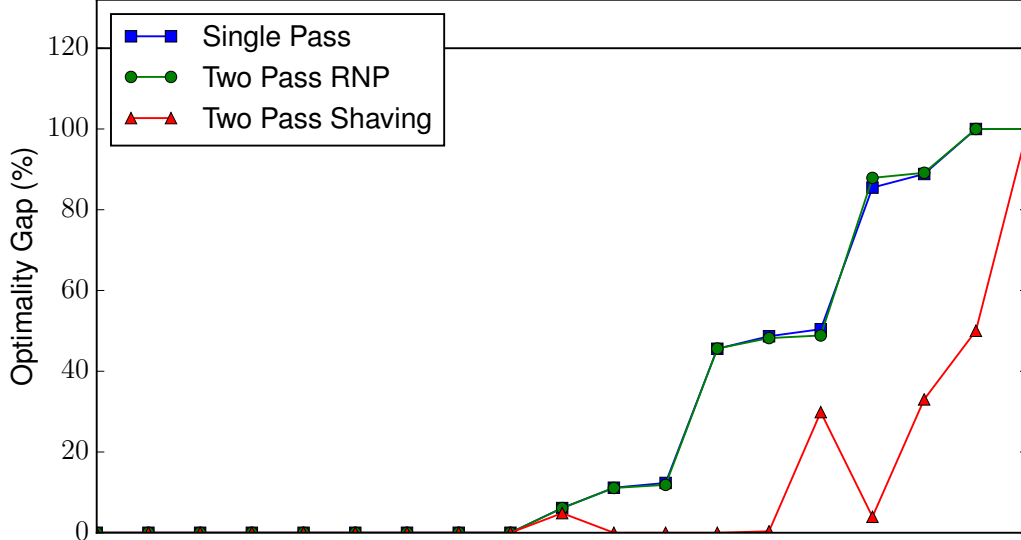


Figure 4.7: Geometric mean optimality gap for the RCMSP problem.

The instances used for the experiment are available from the MiniZinc benchmarks, which are available online¹. The 20 instances used are grouped into *easy*, *medium*, and *hard* classes. Each instance was compiled using three different configurations. The first configuration compiles the instances in a single pass (1P). The second uses a two pass (2P) presolving approach with root-node propagation from Gecode (version 4.3.3). Finally, the third configuration uses two pass presolving with shaving (Sh), which is applied using Gecode to tighten variable bounds before solving. In all three configurations, the resulting FlatZinc was passed to the CPLEX solver (version 12.6). Again, to account for randomness, each instance was solved 6 times using different random seeds. A time limit of 1800 seconds was used, and the memory usage was limited to 4Gb. The experiments were executed on an Intel Core i5 processor clocked at 3.20Ghz with 8Gb of RAM. The solver was instructed to use only one thread for the search.

The results are summarised in Table 4.5. The table shows the size of the resulting programs, in terms of the number of variables and constraints. The mean compilation time and total solve time across the 6 runs is also presented. The geometric mean optimality gap, which is calculated as the best solution found divided by the best objective bound discovered at timeout, is listed here to give some idea of how close the timed out instances were to being solved by the different configurations. The smaller the gap, the closer to an optimal solution the solver was, with 0% indicating a proven optimal solution. For each statistic, the result for a single pass (1P) is shown as is, with the two pass (2P), and two passes with shaving (Sh) results shown as percentages of the single pass result. The best configuration (with a significant lead) for each instance is highlighted in bold.

The tables show that presolving reduced the number of variables and constraints in all cases. Interestingly, shaving always results in more variables than root-node propagation alone. This can be explained by how the new linearisation library (Belov et al., 2016)

¹<https://github.com/MiniZinc/minizinc-benchmarks>

Instance	Variables			Constraints			Compilation		
	1P	2P (%)	Sh (%)	1P	2P (%)	Sh (%)	1P (s)	2P (%)	Sh (%)
easy-1	18116	67.98	78.43	23804	67.62	68.03	1.62	66.77	237.15
easy-2	9328	80.62	86.51	12140	80.29	79.92	0.89	84.62	135.08
easy-3	12047	84.99	88.06	15769	84.80	83.37	1.58	94.63	472.53
easy-4	3783	85.36	89.27	5106	85.59	84.14	0.50	96.00	205.67
easy-5	10196	75.07	84.98	13357	74.75	75.13	0.89	80.90	189.89
easy-6	7130	83.90	90.63	9305	83.69	83.47	0.72	86.57	134.03
medium-1	3790	85.38	89.26	5114	85.61	84.16	0.49	100.68	216.10
medium-2	3330	85.71	90.72	4527	86.02	85.42	0.44	96.60	203.02
medium-3	9020	66.21	75.94	12250	66.96	67.77	1.08	89.83	1478.43
hard-1	59656	64.59	78.17	77434	63.70	63.95	5.23	65.99	546.27
hard-2	79973	63.31	78.32	103977	62.44	62.69	7.37	60.36	935.12
hard-3	39551	69.90	82.13	51506	69.26	69.49	3.63	71.59	371.59
hard-5	5903	84.31	88.34	7818	84.29	82.80	0.76	96.93	282.02
hard-6	9360	85.11	89.51	12253	84.92	83.45	1.18	97.88	400.85
hard-7	15178	65.36	77.64	20435	65.81	66.44	1.88	83.17	2551.73
hard-8	7965	66.38	75.63	10835	67.19	68.05	0.95	86.47	1100.53
hard-9	18505	78.77	87.75	24057	78.33	78.22	1.77	80.43	205.74
hard-10	33532	70.92	83.19	43619	70.28	70.54	3.20	69.90	313.07
hard-11	16271	67.54	79.41	21272	67.01	67.47	1.37	72.35	240.68

	GeoMean Gap			Mean Total Time		
	1P (%)	2P (%)	Sh (%)	1P (s)	2P (%)	Sh (%)
easy-1	0.00	0.00	0.00	110.60	73.56	71.55
easy-2	0.00	0.00	0.00	13.47	98.21	118.47
easy-3	0.00	0.00	0.00	253.92	123.54	137.24
easy-4	0.00	0.00	0.00	34.66	51.44	42.24
easy-5	0.00	0.00	0.00	128.30	126.25	45.79
easy-6	0.00	0.00	0.00	15.51	132.05	105.53
medium-1	0.00	0.00	0.00	55.63	76.70	93.03
medium-2	0.00	0.00	0.00	23.25	92.40	73.01
medium-3	6.12	6.11	4.86	1801.08	99.99	100.83
hard-1	100.00	100.00	50.02	1805.23	99.99	101.29
hard-2	100.00	100.00	100.00	1807.37	99.84	103.41
hard-3	85.48	87.88	3.95	1803.63	99.94	100.55
hard-5	0.00	0.00	0.00	409.79	139.02	33.25
hard-6	12.33	11.86	0.00	1801.18	100.00	4.22
hard-7	88.85	89.15	33.03	1801.88	99.98	102.56
hard-8	50.41	48.86	29.87	1800.95	99.99	100.53
hard-9	11.16	11.09	0.00	1801.77	99.98	45.74
hard-10	48.67	48.21	0.36	1803.20	99.95	100.38
hard-11	45.57	45.62	0.00	1801.37	99.98	7.35

Table 4.5: Path-based two pass presolving comparison for the RCMSP problem.

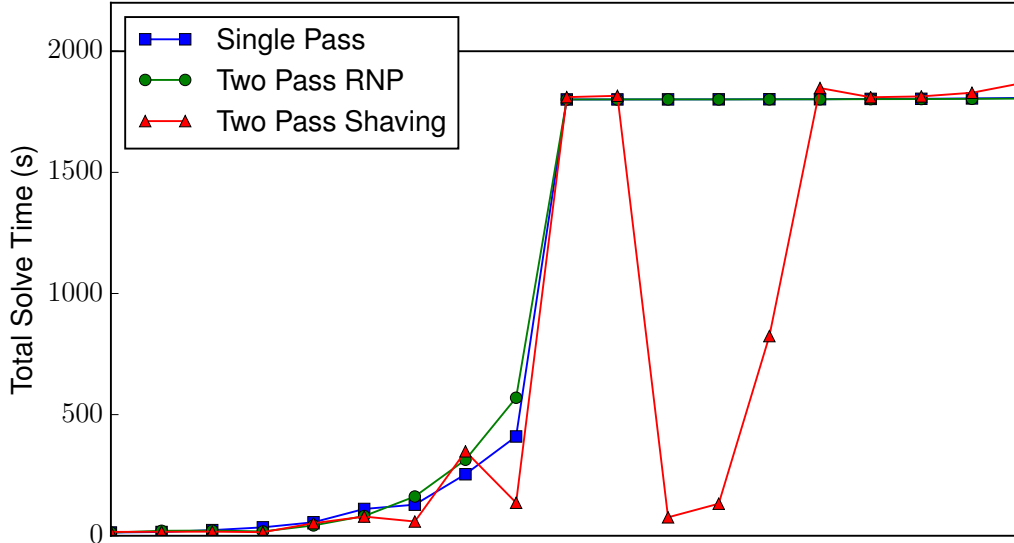


Figure 4.8: Total solve time for the RCMSP problem.

attempts to introduce auxiliary variables for ranges within a domain, rather than for all values. While shaving aims to only tighten the bounds of variables, the propagation that this enables can introduce holes in the domains of variables and, as a result, more ranges need to be represented. The number of constraints resulting from the presolving methods are similar and there is no obvious pattern. Of note however, is the instance **hard-2** which had 103,977 constraints reduced down to only 64,919 (62.44%), an impressive reduction.

Compilation times are affected as expected, with two pass presolving being quite cheap. It reduces the number of variables and constraints required so significantly, that the resulting compilation is faster than with single pass. Shaving, being an expensive technique, increases significantly the compilation time. It is always slower, and sometimes significantly slower than single pass. One instance, **hard-4** has been omitted since its shaving exceeded the memory limit.

For easy instances, the three approaches are all sufficient to solve the problem to optimality in less than 1800 seconds. In the case of the **medium** and **hard** instances, however, a trend emerges. While root node propagation does not appear to remove the hard parts of the programs resulting in a similar performance to single pass compilation, shaving does seem to remove these difficult parts, allowing harder instances (**hard-5**, **hard-6**, **hard-9**, and **hard-11** to be solved within the time limit and obtaining much better optimality gaps. Recall however that, as the mean total time shows, root node propagation can result in faster compile times and can still result in faster total solving.

The optimality gaps and solving times are visualised graphically in Figure 4.7 and Figure 4.8. As in the plots presented earlier, the order of instances is arbitrary, in this case sorted by the mean value for the single pass approach. Figure 4.7 clearly shows that the shaving approach results in better optimality gaps for all but one instance. Figure 4.8 shows that shaving can solve instances much quicker than the single pass and root node

propagation approach. While it is slightly slower in the cases where all runs timeout, this is simply due to the extra cost of performing the shaving during compilation.

This case study has shown that while two pass compilation may reduce the number of variables and constraints in programs, it will not always speed up solving. Experimenting with different presolving configurations can, however, yield huge speed-ups. Shaving, for this specific problem, appears to have a big impact on solve times.

4.6 Conclusion

This chapter had two main contributions. The first, variable paths, presents a method for preserving structure during compilation. The second, path-based multi-pass presolving, presents an improved approach that integrates whole program optimisation with variable paths, allowing useful information, such as tighter domains and variable aliases, to be communicated between several distinct representations of the same instance. Several motivating examples were presented in Section 4.3.1, demonstrating the need for presolving more than just the top-level variables in a compiled program. Further, evaluations of the approach when applied to two different targets was presented showing that, for CP solvers, compiling twice is often too slow to justify the use of multi-pass compilation by default while, for MIP solvers, little compilation overhead is seen with a significant reduction in program size and solving times. A case study is also presented showing how for the RCMSP problem, a multi-pass presolving that makes use of structural information can be beneficial, allowing several hard instances to be solved within a given time limit when stronger techniques are applied during presolving. In conclusion, path-based multi-pass presolving can make the compilation of constraint models more robust, with a low average time overhead, and the potential to improve performance significantly.

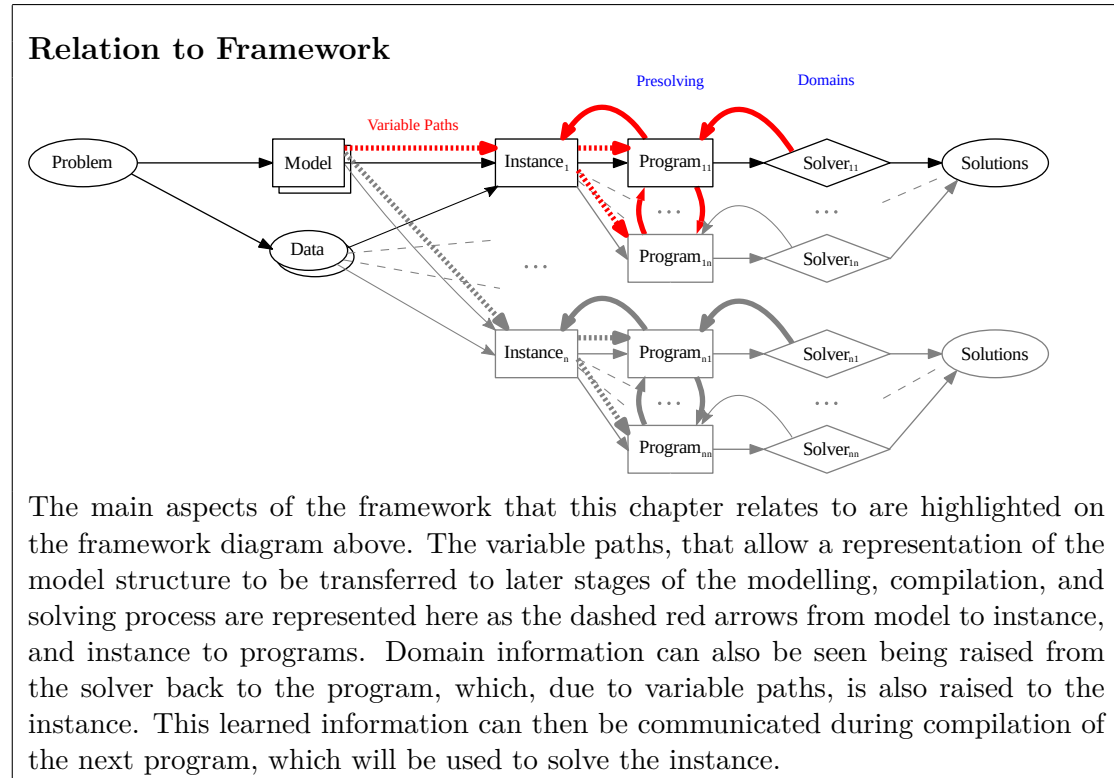
4.6.1 Limitations

This type of presolving may be unnecessary for a CP solver. Most CP solvers will have their own propagation algorithms which should be able to detect when a constraint is entailed and remove it automatically. This motivates the implementation of different passes that may be more beneficial for different kinds of solvers.

In this chapter, only a single form of multi-pass presolving was explored, one where a CP solver is used as the presolver. Other kinds of passes have not been explored yet and may provide interesting results. For example, Dekker (2016) presents a method that can replace a MiniZinc predicate with a `table` constraint by enumerating the possible solutions for it. This could be added as an extra presolving pass in a multi-pass compilation.

4.6.2 Further Research

The contributions of this chapter can be extended and expanded upon in several ways. It may be interesting to explore the benefits of presolving iteratively until a fixed-point is reached and of synchronising additional information such as constraint entailment.



Stronger inference techniques should also be explored, such as supporting arbitrary precision integers and infinities. This could safely and correctly tighten the domains of variables to a point where the target solver can handle instances that previously were out of reach.

There are also several important applications of the new path-based variable synchronisation method that could be explored further. An obvious application of variable paths is in the automated construction of hybrid solvers for arbitrary models. Currently, it is difficult to automatically create a hybrid for a model that communicates on more than just the top-level variables without giving all solvers the same lowest common denominator program. Path-based variable synchronisation would allow for greater communication between solvers.

A small but interesting adjustment to the current approach would be to use a modified presolving library that creates relaxations of a problem for the presolving solver, avoiding the cost of decomposition entirely and focusing the presolving only on the global constraints in a program. This can be achieved by replacing unwanted predicate bodies with `true` in the presolver's library.

Another interesting approach is to merge both the presolving library and the target library together for use by the presolver. This approach would maximise the overlap of introduced variables, improving communication, but it is also likely make compilation more expensive.

Chapter 5

Discovering Structure

5.1 Introduction

As explored in Chapter 3, constraint problems can be modelled in many different ways, and the choice of model can have a significant impact on the efficiency of the resulting programs, that is, on the time that it takes solvers to find (good enough) solutions. Unfortunately, developing good models is often a challenging, iterative process that requires considerable levels of expertise and consumes significant amounts of resources. With this in mind, this chapter aims to answer the second sub-goal presented in Chapter 1:

Can the explicit structure of a model guide the search for and exposure of implicit structure?

To answer this question, the chapter presents an approach that supports users through this iterative process by suggesting possible model improvements based on the detection of implicit model structures, such as global constraints.

Motivation

Using global constraints in a model has three significant advantages. The first advantage is improved efficiency when using one of the many solvers that implement specialised propagators for global constraints. In such cases, having the global constraint appear explicitly in the model can considerably improve the efficiency of the solving process for all instances of the model. The second advantage is improved efficiency even when using solvers that do not implement the global constraints. This is because the presence of global constraints in the model provides information regarding the implicit structure of the model that might not have been apparent without them. This information can also be used to improve the effectiveness of the resulting constraint program. For example, global constraints can facilitate the automatic detection of symmetries (Van Hentenryck et al., 2005), which can then be exploited either by adding symmetry breaking constraints or by modifying the search. Also, as discussed in Chapter 4, the presence of a global constraint in the model can be used by the compiler to make better decisions during compilation, by tightening variable bounds and selecting appropriate decompositions when a target solver

does not support the inferred global constraint. The third advantage is the fact that global constraints might make the model more readable to modellers.

Unfortunately, this might be difficult for modellers. For example, naive modellers might simply be unaware of global constraints, while expert modellers might be unfamiliar with the globals available for a specific modelling language. Further, globals might only apply to alternative (or *dual*) model viewpoints (Law et al., 2002) of which the modeller may not be aware (e.g., zero-one variables that could explicitly represent integer variables). Such viewpoints can also be seen as implicit structures and, if made explicit, can in turn enable the detection of other implicit structures. As a result, techniques able to detect implicit structures in a model are useful.

Contribution

This chapter presents a method, implemented by the MiniZinc Globalizer, which **uses explicit model structure to guide the discovery of implicit structure that can be made explicit**. In particular, given a model for a constraint problem and several input data files, the method suggests global constraints as possible replacements for combinations of simple constraints from the model. This method can be applied iteratively, allowing the system to expose, for example, alternative viewpoints (Cheng et al., 1999) of the variables in the model which, when represented as global constraints and made explicit, enable the detection of other implicit structures. This helps users write higher level models that can be solved more efficiently, can be compiled better (see Chapter 4), and can improve the modeller’s understanding of the problem.

The method exploits both the explicit and implicit structure of the model by considering combinations of the constraints, collections of variables, parameters and loops already present in the model, as well as parameter data from several data files. Each candidate global constraint is assigned a score by comparing a sample of its solution space with that of the part of the model it is intended to replace. The top-scoring candidates are presented to the user through an interactive display, which shows how they could be incorporated into the model.

This method has many novel characteristics when compared to other automatic constraint model inference and transformation methods (e.g., Beldiceanu et al. (2011, 2012), Bessiere et al. (2007), Charnley et al. (2006b), Frisch et al. (2001, 2003, 2005), and Gent et al. (2007a); see Section 5.3 for a detailed discussion). First, most other methods focus on using the known solutions and non-solutions of a constraint problem to infer the constraints for an instance of a problem. Instead, our approach splits an already complete and correct model, along with several data files, into submodels. The candidate global constraint is then allowed to be directly associated with the group of constraints it replaces (those in the submodel). Second, the generation of arguments for the candidate global constraints uses the variables, parameters and collections of variables appearing in the associated submodel. Likely arguments for constraints can then be generated more efficiently. Further, it means the candidate global constraints are defined at the model level rather than at the instance level. This is important not only for the user, but also

for the third novel characteristic: the method uses the solutions from different instances (rather than those from a single one) to generate, rank and filter the candidates. This increases its accuracy considerably (as shown experimentally in Section 5.6).

The resulting tool – the *MiniZinc Globalizer* – was originally made available through a web interface however this service is no longer available. A new version that is integrated with the MiniZinc IDE and allows off-line use is in development and a release is planned. This version can be more easily integrated into a user’s typical modelling workflow.

Structure of the Chapter

The rest of the chapter is organised as follows. Section 5.2 introduces a model that will be used as a running example throughout the chapter, together with some common global constraints that have not been discussed thus far. Section 5.3 discusses related work. Section 5.4 introduces the proposed method to globalize a constraint model and later extends it to be able to infer relationships among hidden structures by representing these hidden structures as constraints to be detected in an initial pass. Section 5.5 provides some details regarding the implementation of the method used in this chapter to perform the experiments reported in Section 5.6, which show that the method can detect these hidden structures and, in addition, is not prohibitively slow. Finally, Section 5.7 presents the conclusions.

5.2 The Progressive Party Problem

A version of the *Progressive Party Problem* will be used as a running example in this chapter. The goal in solving this problem is to organise a party between the crews of several boats at a yacht club. The rules of the party state that only certain boats can host visitors, while the crews of the remaining boats must take turns visiting these host boats in different time periods. Further, the host boats have capacities that limit the number of guests they can accommodate, each guest crew can only visit a host once, and guest crews cannot meet more than once.

The first known model for this problem was presented in Smith et al. (1996), which compared a zero-one integer program (a specialisation of MIP, involving only zero-one variables) for this problem against a CSP. As seen previously, MIP formulations often require many more constraints to describe an instance than a CP formulation. The integer program suffered from this issue and, as a result, had too many constraints, while a second formulation was able to successfully find a 13-host solution.

A MiniZinc version of the second model is shown in Listing 5.1. Lines 1–6 declare the parameters of the model. Line 1 declares the first three parameters: the number of time periods `p`, number of host boats `nh`, and number of guest crews `ng`. These three parameters are used in lines 2–4 to declare the sets of designated host boats `HostBoats`, guest crews `GuestCrews` and time periods `Time`, where each host boat, guest crew and time period is identified by a distinct integer in the ranges `1..nh`, `1..ng` and `1..p`. These sets are used in lines 5 and 6 to declare two arrays of parameters representing the number of

```

1  int: p; int: nh; int: ng;
2  set of int : HostBoats = 1..nh;
3  set of int : GuestCrews = 1..ng;
4  set of int : Time = 1..p;
5  array [GuestCrews] of int : crew;
6  array [HostBoats] of int : capacity;
7
8  array [GuestCrews, Time] of var HostBoats : hostedBy;
9  array [GuestCrews, HostBoats, Time] of var 0..1 : visits;
10 constraint forall (g in GuestCrews, h in HostBoats, t in Time)
11     ( visits[g,h,t] = 1 <-> hostedBy[g,t]=h ); % channel
12
13 constraint forall (h in HostBoats)
14     ( forall (g in GuestCrews)
15         (sum (t in Time) (visits[g,h,t]) <= 1) % at_most_one_visit
16         /\ forall (t in Time)
17             (sum (g in GuestCrews) (crew[g]*visits[g,h,t]) <= capacity[h])); % capacity
18
19
20 array [GuestCrews, GuestCrews, Time] of var 0..1 : meet;
21 constraint forall (k, l in GuestCrews where k<l)
22     ( forall (t in Time)
23         (hostedBy[k,t] = hostedBy[l,t] -> meet[k,l,t] = 1) % will_meet
24         /\ sum (t in Time) (meet[k,l,t]) <= 1 ); % meet_once
25
26 solve satisfy;
27 output [show(hostedBy)];

```

Listing 5.1: MiniZinc model for the Progressive Party Problem based on the second formulation proposed in Smith et al. (1996).

members in each guest crew and the maximum capacity of each boat. The values for all these parameters are provided in data files. The following is a data file for the model:

```

% party.1.dzn
crew = [6, 5, 4, 4, 4, 3, 2, 2, 2, 2];
capacity = [7, 10, 8, 8, 11];

nh = 5;
ng = 10;
p = 3;

```

The main decision variables in the model are declared in line 8 by a two dimensional array, where the variable selected by the array access `hostedBy[g,t]` represents the boat in `HostBoats` that hosts guest crew `g` at time `t`, with `g` coming from the set `GuestCrews`, and `t` coming from the set `Time`. Lines 9–11 declare a three dimensional array of auxiliary zero-one variables satisfying a constraint (labelled `channel`) that ensures auxiliary variable `visits[g,h,t]` is 1 if and only if `hostedBy[g,t]=h`. That is, it is 1 if guest crew `g` visits host boat `h` at time `t`. These auxiliary variables are used in lines 13–17 to (a) constrain each guest crew `g` to visit each host boat `h` at most once (by summing up the value of `visits[g,h,t]` for every time period `t`, and ensuring the sum is less or equal to 1); and (b) ensure the capacity constraints are satisfied (by summing up the members `crew[g]` of any guest crew `g` that visits the same host boat `h` at the same time `t`, and ensuring

that the sum is less than or equal to `capacity[h]`). Note that the constraint (a) is set by lines 13–15 (labelled `at_most_one_visit`), while constraint (b) is set by line 13 (labelled `capacity`) together with lines 16–17. This is achieved by using conjunction (represented by the symbol \wedge) and nested forall loops to build a *conjoined constraint*. This conjoined constraint could easily be replaced by two constraints, by repeating line 13 right before line 16 (and appropriately dealing with parenthesis and semicolons). While this might be considered simpler, it might also be considered less structured and, thus, less clear to readers, as they would be required to notice that both loops use the same index sets.

Finally, lines 20–24 introduce another three dimensional array of auxiliary zero-one variables with a constraint (labelled `will_meet`) that ensures that the auxiliary variable `meet[k,l,t]` is 1 if and only if `hostedBy[k,t]=hostedBy[l,t]`. That is, it is 1 if guest crews `k` and `l` met at time period `t`. These auxiliary variables are then used to constrain each pair of guest crews to meet at most once (labelled `meet_once`). Again, the constraints are conjoined and nested within the outer forall set in line 20.

As mentioned in Section 2.3.4, a global constraint establishes a relation between a non-fixed number of variables. We have already discussed the well-known `alldifferent` constraint. We will now introduce another common class of global constraint that will be used in the rest of this chapter: *channelling constraints* (Cheng et al., 1996), which are often added to establish a bijective connection between two different arrays of variables. This is useful when, for example, the problem can be modelled using different types of variables that represent the same information redundantly. In such cases, it is often advantageous to create the two kinds of variables, use them to define the different (possibly redundant) constraints, and then add a channel constraint to connect the two types of variables (and, thus, their associated constraints). This can be advantageous either because some constraints are easier to express when using a given type of variable, or because the aim is to create two alternative models (often referred to as duals) that can benefit from each other's propagation by means of the channelling constraints. There are two main kinds of channelling constraints. The first one is `inverse(XVars,YVars)` which connects the two arrays of integers variables `XVars` and `YVars` as follows: $XVars[i] = j \leftrightarrow YVars[j] = i$. This constraint is often used when modelling problems involving successor and predecessor variables, with `XVars[i]` indicating the task that will follow task `i` and `YVars[j]` indicating the task that should precede task `j`. The second one is `channel(x,BVars)`, which connects an integer variable `x` with an array of zero-one variables `BVars` as follows: $x = j \leftrightarrow BVars[j] = 1$. As explored in Section 2.3.2 and Section 2.3.3, some constraints sometimes require a binary variable *view* of some integer variables, and vice-versa. The `channel` constraint encodes the link between the integer and binary view of the variables.

Another constraint of interest for this Chapter is the `bin-packing-capacity(C,x,w)` global constraint. This is a variant of the `bin-packing` constraint described in Section 2.3.4, where objects in the array `x` with weights given by array `w` are packed in bins, such that the capacities of the bins given by array `C` are not violated. More formally, each object has a weight w_i and each bin has a capacity C_j , and the variable x_i is assigned to value j if object i is packed in bin j . The global constraint can be decomposed as $\forall j : C_j \geq \sum_i w_i (x_i = j)$.

Bin-packing has found use in several fields, ranging from loading cargo into delivery trucks to stock-cutting problems (Coffman et al., 1997).

An expert modeller can immediately see that line 15 expresses an `alldifferent` global constraint on the `hostedBy` variables for each `g` **in** `GuestCrews`, which would be written as `alldifferent([hostedBy[g,t] | t in Time])`. This can be seen by examining the `channel` constraint which connects `visits` to the `hostedBy` variables. Since for a fixed guest and host only one `visits` variable will be set to 1, a host can only be assigned for a fixed guest and time in the `hostedBy` variables once. It is less obvious that line 17 can be expressed with the `bin-packing-capacity` constraint with arguments: `capacity`, `[hostedBy[g,t] | g in GuestCrews]`, and `crew`. The `capacity` constraints encode this with the reification of $x_i = j$ being provided by the `channel` constraints. The goal then, is to automatically detect these global constraints so that a user may add them to their model.

To simplify the discussion of the running example, the remaining sections will use the following shorthand notation to express the main structure of the above model: $(\forall GHT : \text{channel}) \wedge (\forall H : (\forall G : \text{at_most_one_visit}) \wedge (\forall T : \text{capacity})) \wedge (\forall GG : (\forall T : \text{will_meet}) \wedge \text{meet_once})$, where *channel* denotes the constraint appearing in lines 10–11, *at_most_one_visit* that in line 15, *capacity* that in line 17, *will_meet* that in line 23, and *meet_once* that in line 24. Universal quantifications over G , H and T correspond to loops over the sets `GuestCrews`, `HostBoats`, and `Time`, respectively. We call G, H and T the *index sets* of their loops. For simplicity, we always write nested `forall` loops using a single quantifier and disregard the order of their index sets, e.g., $\forall GHT$ is equivalent to $\forall T \forall GH$, to $\forall H \forall T \forall G$, and so on.

5.3 Related Work

There are two main lines of research related to this work: *constraint acquisition* and *automatic model transformation*. In the *acquisition* line of work, the closest works are to be found in Beldiceanu et al. (2011, 2012), which introduced Constraint Seeker to infer global constraints from positive and negative examples of solutions, and Model Seeker to infer an entire constraint program (i.e., a concrete set of variables and constraints) given a set of complete solutions to a constraint problem. Note, that it is common in the constraint programming literature to refer to a constraint program as a “model”, but in this thesis we distinguish between parametric models and concrete programs. Constraint Seeker and Model Seeker differ from the method presented here both in motivation and methodology. The motivation of this chapter is to identify parts of a given model that can be replaced by global constraints. Having access to an initial model significantly affects the methodology, as it allows us to make extensive use of the explicit information contained in the model. In particular, it allows us to (a) focus on submodels that are equivalent to a single global constraint, as opposed to a conjunction of them, (b) significantly reduce the search for possible combinations of global constraint arguments, while increasing the likelihood of obtaining meaningful ones, and (c) consider not only the solution variables, but any other intermediate variables in the model and its input data. Having the input data also affects the methodology, as it allows us to (a) better generate candidates and (b)

automatically generate as many solutions as required for ranking. For example, the input data enables us to derive **bin-packing** constraints for the Progressive Party problem, while Model Seeker cannot infer these from the solutions alone, since the constraints depend on fixed parameters for the weights and capacities that are only available in the input data.

The method presented here also relates to the CGRASS system (Frisch et al., 2001, 2003), which among other model transformations, syntactically matches cliques of disequality constraints that can be replaced by an **alldifferent** constraint. This requires the modeller to implement the constraint using the exact structure to be matched. The approach taken in this chapter goes beyond this, to infer any of a set of global constraints using a general (rather than specialised) method, and does so for a model, rather than for single instances. Syntactic matching could be used however to catch common formulations.

Other acquisition approaches focus on the automatic generation of implied constraints. A general method is described in Charnley et al. (2006b), where machine learning is used to induce constraints for the solutions of small problems, and a theorem prover is then used to show the constraints hold for the model. The generality of the method results in applicability restrictions: the model can only be parameterised by a single integer, and it needs to be expressible in first order logic. In our case the constraints are already predetermined (the list of global constraints considered) and, thus, the data can be as complex as necessary. Further, we do not attempt to prove the correctness of the constraints as this reduces to proving the equivalence of two models, which is undecidable.

Another related method is that of CONACQ (Bessiere et al., 2005a) which, given examples of solutions and non-solutions for a target problem and a library of constraints, *acquires* constraint networks, that is, conjunctions of constraints in the library that are consistent given solutions and non-solutions. It uses the SAT-based *version space algorithm*, where the version space is the set of all constraint networks defined from the library that are consistent with the examples. While general and powerful, it considers instances of models, rather than models themselves. Further, it relies on the library of constraints being relatively small. This is not the case for the approach presented in this chapter.

Finally, (Bessiere et al., 2007) describes how implied parametric constraints can be learned by first adding a large disjunction of constraints with different parameters that together are trivially implied, and then successively pruning that disjunction by removing constraints that do not change the solutions found. This method goes further than what we attempt in that it infers parameters from solutions, while we only try to match parameters that are already given in the model. For functional dependencies, however, we can infer parameter sets, as in the **Schedule** example in Section 5.6.

In the area of *model transformation*, the work on ESSENCE' (Frisch et al., 2005; Gent et al., 2007a) is somewhat related. These systems transform a model specified in a highly abstract manner into a more concrete one. The method in this chapter moves in the opposite direction: we detect parts of a model that are instances of a more generic model pattern. While this generic pattern is currently restricted to global constraints, it is straightforward to extend the method to use any other useful constraint pattern (see Section 5.4.5). In fact, globalization and automatic transformation are complementary: starting from a

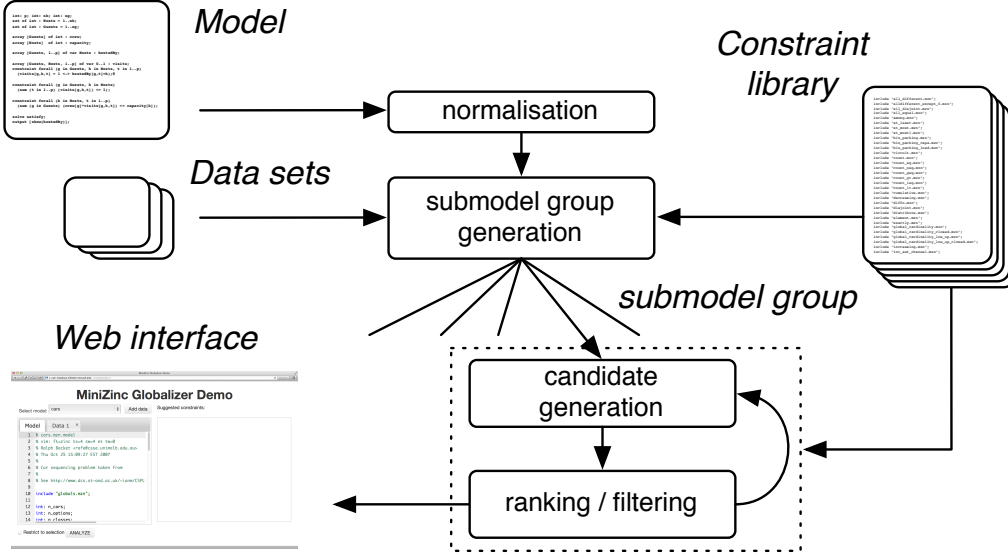


Figure 5.1: Graphical overview of model globalization.

Algorithm 5.1 High-level algorithm for Globalizer.

```

procedure globalize( $M, DtFiles, Lib, SubModelSize$ )
   $\langle Constrs, Decls \rangle \leftarrow \text{normalize}(M)$ 
   $Submodels \leftarrow \text{generate\_submodels}(\langle Constrs, Decls \rangle, SubModelSize)$ 
   $Candidates \leftarrow \emptyset$ 
  for each  $Submodel \in Submodels$  do
     $SubmodelInstanceGroup \leftarrow \text{instantiate\_submodel}(Submodel, DtFiles)$ 
     $Candidates \leftarrow Candidates \cup \text{process\_group}(SubmodelInstanceGroup, Lib)$ 
  return  $Candidates$ 

```

low-level model, globalization yields a high-level model that is then amenable to automatic transformation.

5.4 Globalization

Figure 5.1 provides a graphical view of the main steps of the method, which can also be seen in high-level algorithmic form in Algorithm 5.1. Intuitively, the method starts by *normalising* the input model M , that is, by separating the declarations $Decls$ from the constraints $Constrs$, and normalising the constraints in such a way that conjoined constraints are split into their component constraints. After normalization, several *Submodels* are generated as targets for globalisation, where each such target $Submodel \in Submodels$ corresponds to the submodel formed by combining $Decls$ with a particular subset of the normalised constraints in $Constrs$, possibly unrolled. The size of the submodel is controlled by the input parameter $SubModelSize$, which indicates the number of constraints that will appear in the submodel. Each such submodel is then instantiated with the data provided by each data file appearing in the set $DtFiles$, obtaining a group of submodel instances. Each such group is then processed to determine its associated set of *candidate global constraints*. To achieve this, the method generates an initial set of candidates called $Candidates$ from those present in the constraint library Lib , where each global constraint

in this set is a candidate for equivalence to the submodel instance group associated to *Submodel*. These generated candidates are then processed in order to *score* them according to how well their solution space matches that of the submodel instance group, and *filter them out* if their score is below a given threshold. Note that the instantiation of the submodel is needed to be able to find its solution space and test its equivalence with that of the global constraint. Finally, the filtered candidates are returned and, as shown in Section 5.5.3, they are presented to the user by means of an interactive GUI. The following sections discuss each of these steps in detail.

5.4.1 Normalization

The algorithm for normalizing a model M is simple and is shown in Algorithm 5.2. The procedure partitions M into two sets: the set *Constrs* of *normalized* constraints and the set *Decls* of original variable and parameter declarations. The constraints in the model are normalized by exhaustively applying two simple rewriting rules that (a) split simple conjunctions of the form $c_1 \wedge \dots \wedge c_n$, into their individual constraints c_1, \dots, c_n , and (b) split *forall* loops that contain conjunctions into individual *forall* loops. For example, in the case of the Progressive Party model, *Decls* will be initialised as the set containing all declarations in the model, that is, all declarations appearing in lines 1–9 plus the one in line 20. *SC* will be initialised as the set containing the three top-level constraints in the model, that is, $\{\forall GHT : channel, \forall H : (\forall G : at_most_one_visit) \wedge (\forall T : capacity), \forall GG : (\forall T : will_meet) \wedge meet_once\}$. After normalizing the Progressive Party model, *Constrs* will contain the following five constraints $\{\forall GHT : channel, \forall GH : at_most_one_visit, \forall HT : capacity, \forall GGT : will_meet, \forall GG : meet_once\}$.

Normalization is vital for discovering global constraints that describe parts of a constraint, rather than all of it. For example, the combination of the constraint $\forall GH : at_most_one_visit$ with the channelling constraint $\forall GHT : channel$ in the Progressive Party is equivalent to a conjunction of several *alldifferent* constraints. To discover this, each component constraint must be considered separately, so that they can be combined.

Algorithm 5.2 Flatten conjunctions and aggregate *forall* quantifier to normalize.

```

procedure normalize( $M$ )
   $Decls \leftarrow$  set of all variable and parameter declarations in  $M$ 
   $Constrs \leftarrow$  set of constraints in  $M$ 
  while one of the following rules applies do
    if there is a  $c \in Constrs$  of the form  $(c_1 \wedge \dots \wedge c_n)$  then
       $Constrs \leftarrow (Constrs \setminus c) \cup \{c_1, \dots, c_n\}$ 
    if there is a  $c \in Constrs$  of the form  $(\forall A_1 \dots \forall A_n : c_1 \wedge \dots \wedge c_m)$  then
       $Constrs \leftarrow (Constrs \setminus c) \cup \{\forall A_1 \dots \forall A_n : c_1, \dots, \forall A_1 \dots \forall A_n : c_m\}$ 
  return  $\langle Constrs, Decls \rangle$ 

```

Algorithm 5.3 Splitting a set *Constrs* of constraints into submodels.

```

procedure generate_submodels( $\langle \text{Constrs}, \text{Decls} \rangle$ , SubModelSize)
  targetSet  $\leftarrow \emptyset$ 
  for each ( $S \subseteq \text{Constrs}$  where  $1 \leq |S| \leq \text{SubModelSize}$ ) do
    Decls'  $\leftarrow$  subset of Decls that includes variables and parameters of S
    targetSet  $\leftarrow \text{targetSet} \cup \{ \langle S, \text{Decls}' \rangle \} \cup \text{unrollings}(\langle S, \text{Decls}' \rangle)$ 
  return targetSet

procedure unrollings( $\langle \text{Constrs}, \text{Decls} \rangle$ )
  submodels  $\leftarrow \emptyset$ 
  if Constrs are of the form  $(\forall a_1 \in A_1, \forall a_2 \in A_2, \dots \forall a_n \in A_n \langle \text{where } c \rangle) : e$  then
    for each ( $X \subseteq \{a_1 \in A_1, a_2 \in A_2, \dots a_n \in A_n\}$ ) do
      if X is not empty then
        if Constrs has  $\langle \text{where } c \rangle$  that depends on X then
          Constrs'  $\leftarrow (c \rightarrow \text{Constrs})$  with X removed from the forall
        else
          Constrs'  $\leftarrow$  Constrs with X removed from the forall
        for each ( $a_i \in A_i$  in X) do
          Constrs'  $\leftarrow \text{Constrs}' \cup \{a_i = \text{val}(A_i)\}$ 
        submodels  $\leftarrow \text{submodels} \cup \{ \langle \text{Constrs}', \text{Decls} \rangle \}$ 
  return submodels

```

5.4.2 Generating Submodels

After normalizing a given model *M*, the next step is to generate the submodels of *M* that will be considered as targets for globalization. The goal here is to find many combinations of constraints from the model for which we may be able to find an equivalent global constraint. Algorithm 5.3 shows how this is achieved by the `generate_submodels` procedure, which takes as arguments the full set of constraints (*Constrs*) and the number of constraints to include in each submodel (*SubModelSize*), and generates all submodels that can be formed by conjunctions of the given number of possibly unrolled constraints in the model (in our implementation we default to using two constraints). The procedure works as follows. For each subset $S \in \text{Constrs}$ of appropriate size, it adds *S* and the possible unrollings of *S* (`unrollings(S)`) to the set *targetSet*, which will later be returned with the full set of generated submodels. Listing 5.2 shows an example submodel that corresponds to the $\forall GHT : \text{channel}$ constraint from the model. Each submodel *S* will be unrolled by the `unrollings` procedure only if its constraints are of the form $(\forall a_1 \in A_1, \forall a_2 \in A_2, \dots \forall a_n \in A_n \langle \text{where } c \rangle) : e$, that is, if they are defined over a (possibly nested) universally quantified expression with an optional `where` condition *c*. If so, unrolling is achieved by selecting some subset *X* of the quantified variables and constructing a new set of constraints *Constrs'* by removing each of these variables $a_i \in X$ from the quantified part and equating a_i to expression $\text{val}(A_i)$, denoting a value in domain

```

constraint forall (g in GuestCrews, h in HostBoats, t in Time)
  ( visits[g,h,t] = 1 <-> hostledBy[g,t]=h );

```

Listing 5.2: Example: generated submodel.

```

% Submodel S with two constraint items, as they appear in the model
constraint forall(h in HostBoats, t in Time)
  (sum(g in GuestCrews) (crew[g] * visits[g, h, t]) <= capacity[h]);
constraint forall(k,l in GuestCrews, t in Time where k < l)
  (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% Unrolling of S with t fixed and removed from the two forall loops
t = val(Times);
constraint forall(h in HostBoats)
  (sum(g in GuestCrews) (crew[g] * visits[g, h, t]) <= capacity[h])
constraint forall(k,l in GuestCrews where k < l)
  (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% Submodel S' with a single constraint item
constraint forall(k,l in GuestCrews where k < l)
  (hostedBy[k, t] = hostedBy[l, t] -> meet[k, l, t] = 1);

% Unrolling of S' with l fixed and removed
l = val(GuestCrews)
constraint forall(k in GuestCrews where k < l)
  (sum(t in Time) (meet[k, l, t]) <= 1);

% Unrolling of S' with k fixed and removed
k = val(GuestCrews);
constraint forall(l in GuestCrews where k < l)
  (sum(t in Time) (meet[k, l, t]) <= 1);

% Unrolling of S' with k and l fixed and removed. Note that the
% condition from the forall now implies the constraint.
k = val(GuestCrews); l = val(GuestCrews);
constraint k < l -> sum(t in Time) (meet[k, l, t]) <= 1;

...

```

Listing 5.3: Selection of submodels showing loop unrolling.

set A_i . These values will later be fixed to an arbitrary value (in our concrete case, the minimum, maximum, or middle value) of their domain. This is achieved by adding the relevant $a_i = \text{val}(A_i)$ constraints to each submodel *Constrs'*. If the condition c depends on the quantified variables in X it is moved from the *where* into the constraint e as an implication. Listing 5.4 shows a partial unrolling of the $\forall GHT : \text{channel}$ constraint.

For the example model in Listing 5.1, the algorithm generates many possible target submodels in addition to the one presented in Listing 5.4. More examples of partial unrollings of submodels are presented in Listing 5.3 (separated by empty lines). These examples show how the instantiation of $\text{val}(A_i)$ works. The first submodel has two constraint items which correspond to a particular target S in the algorithm. The second submodel corresponds to an unrolling of S , where the value for t has been fixed and $t \text{ in Time}$ has been removed from the *forall* loops of both constraints. The third submodel has a single constraint on

```

int: g = val(GuestCrews);
constraint forall (h in HostBoats, t in Time)
  ( visits[g,h,t] = 1 <-> hostedBy[g,t]=h );

```

Listing 5.4: Example: partial unrolling.

Algorithm 5.4 Instantiating a set of submodels.

```

procedure instantiate_submodel( $\langle Constrs, Decls \rangle, DtFiles$ )
  for each  $DtFile \in DtFiles$  do
     $Decls \leftarrow Decls \cup DtFile$   $\triangleright$  Assign parameters with  $DtFile$ 
     $TempSetDecls \leftarrow \{Decls\}$ 
    for each declaration of the form  $x = val(A_x) \in Decls$  do
      for each  $TempDecl \in TempSetDecls$  do
         $TempSetDecls \leftarrow TempSetDecls \setminus TempDecl$ 
        for each value  $v \in \{min(A_x), max(A_x), mid(A_x)\}$  do
           $TempSetDecls \leftarrow TempSetDecls \cup (TempDecl \setminus x = val(A_x) \cup \{x = v\})$ 
     $SubmodelInstanceGroup \leftarrow \emptyset$ 
    for each  $TempDecl \in TempSetDecls$  do
      if  $\forall c_1, c_2 \in Constrs, c_1$  and  $c_2$  are connected then
         $SubmodelInstanceGroup \leftarrow SubmodelInstanceGroup \cup \{\langle Constrs, TempDecl \rangle\}$ 
  return  $SubmodelInstanceGroup$ 

```

its own, representing another target S' in the algorithm. The next two submodels unroll S' by fixing variables k and l , and removing them from their respective targets. The final submodel also unrolls S' and shows what happens when all loop variables are fixed and there is a `where` condition ($k < l$) that must be considered: the condition implies the remaining constraint. In this way, the implication ($k < l \rightarrow \dots$) ensures that candidates cannot be compared against this submodel when the condition is not met.

5.4.3 Instantiating Submodels

Each submodel generated in the previous step will give rise to a group of concrete instances, constructed by instantiating the unrolled variables with concrete values and by adding given data values for the model's parameters.

This is achieved by the procedure presented in Algorithm 5.4, which takes as input a submodel (represented by its declarations $Decls$ and its constraints $Constrs$) and a set of data files $DtFiles$, and proceeds as follows. It first constructs in $TempSetDecls$ all possible instantiations of the declarations $Decls$ using both the different input data files and some of the values of the unrolled variables. As discussed later in Section 5.5.1, while we have used $min(x)$, $max(x)$, and $mid(x)$, representing the minimum, maximum and mid value, respectively, of the domain associated to quantified variable x , any other concrete values could have been used. Listing 5.5 shows the example submodel after being partially unrolled and instantiated with g being fixed to the minimum value from the set `GuestCrews`. After computing $TempSetDecls$, each of its elements is combined with the constraints $Constrs$ in the submodel to produce a submodel instance. Note that this

```

int: g = min(GuestCrews);
constraint forall (h in HostBoats, t in Time)
  ( visits[g,h,t] = 1 <-> hostedBy[g,t]=h );

```

Listing 5.5: Example: unrolled subinstance.

last step only happens if all pairs of constraints in the resulting instance are *connected*, that is, if the constraints either directly share at least one uninstantiated variable, or there is a path of connecting constraints for them in *Constrs*, such that every two adjoint constraints in the path are connected. This is a reasonable optimisation as constraints represent a connection between variables and, if there is no connection in the model, one cannot possibly be inferred.

5.4.4 Processing Submodel Instance Groups

The procedure `instantiate_submodel(Submodel, DtFiles)` returns in *SubmodelInstanceGroup* a set containing different instances of *Submodel* constructed using *DtFiles*. Recall that each such instance has different parameter values due either to different data files given by the user, or to the different values chosen (from $\min(x)$, $\max(x)$, and $\text{mid}(x)$) during loop unrolling. For each *Submodel* of the original model *M*, its associated *SubmodelInstanceGroup* is then processed in order to generate, rank and filter a set of candidate constraints, which are then added to the final set *Candidates* for *M*. This is achieved by the procedure `process_group` (shown in Algorithm 5.5) which, in addition to *SubmodelInstanceGroup*, takes the declarations *Decls* in the original model and the constraint library *Lib* as input parameters, as they are needed to generate the candidates.

Processing all instances of each submodel as a group allows `process_group` to increase its accuracy by taking the intersection of the candidate constraints found for each *SubModelInstance* in *SubmodelInstanceGroup*. This is correct because a global constraint can only be equivalent to a submodel (and, thus, be a candidate) if it holds for every instantiation of the submodel. Initially, the full set of constraints and argument tuples (written as the special symbol *Universe*) is considered as the set of possible candidates. Thanks to the filtering (intersection) performed later in the process, the number of candidates in this initial set may decrease with each subsequent instance in the group, and only those that remain after processing the final instance are returned as the result. These remaining candidates are exactly the intersection we seek to compute.

Algorithm 5.5 Processing submodel instance groups.

```

procedure process_group(SubmodelInstanceGroup, Decls, Lib)
  Candidates  $\leftarrow$  Universe
  for each SubModelInstance in SubmodelInstanceGroup do
    Candidates  $\leftarrow$  Candidates  $\cap$  generate_candidates(SubModelInstance, Decls, Lib)
    Candidates  $\leftarrow$  rank_and_filter(SubModelInstance, Candidates)
  return Candidates

```

Candidate generation

The first step of the group processing is the generation of candidate constraints for each *SubmodelInstance* in *SubmodelInstanceGroup*. The algorithm for generating candidates is shown in Algorithm 5.6. It takes as input a *SubmodelInstance* in its associate *SubmodelInstanceGroup*, the set *Decls* of declarations of the original model *M*, and the

Algorithm 5.6 Generating candidate constraints for a submodel instance.

```

procedure generate_candidates(SubModelInstance, Decls, Lib)
  Candidates  $\leftarrow \emptyset$ 
  Solutions  $\leftarrow$  disjunction of random sample of solutions of SubModelInstance
  Template  $\leftarrow (SubModelInstance \setminus Decls) \cup Solutions$ 
  BaseArguments  $\leftarrow$ 
    (variable and parameter collections in SubModelInstance)  $\cup$ 
    (variable and parameter sub-collections in constraints of SubModelInstance)
  Arguments  $\leftarrow BaseArguments \cup$ 
    (array accesses of elements of BaseArguments)  $\cup$ 
    { constant 0 }  $\cup$  { blank symbol }
  for each constraint cons in Lib do
    for each tuple args that can be built from Arguments do
      Replace blank symbols in args by their value if necessary
      Instance  $\leftarrow Template \cup (\text{constraints for } cons(args))$ 
      if Instance is satisfiable then
        Candidates  $\leftarrow Candidates \cup \{cons(args)\}$ 
  return Candidates

```

library of global constraints *Lib*. Note that each constraint entry in *Lib* has a signature comprising the name, arity, and valid argument types. In addition, arguments can have associated information indicating whether they are functionally dependent on other arguments, and stating conditions that must be met for the argument to be used. See Section 5.5.2 for details on the particular *Lib* used by the implementation.

The algorithm proceeds as follows. First it finds a reasonably large random sample of solutions of the given *SubModelInstance* (the default number of samples is 30, see Section 5.6 for details), and constructs *Solutions* as their disjunction. Intuitively, we will consider a global constraint as a candidate for the submodel if it is satisfied by *all* these sample *Solutions*. To determine this, we create a template submodel in such a way that, if after adding the global constraint, the template submodel remains satisfiable, then the global constraint satisfies *all* sample *Solutions* and can be considered a candidate. This template submodel is built by adding the *Solutions* to the parameters and variables in *Decls*. Note that this template is trivial to satisfy.

Possible candidate constraints are then obtained by combining each global constraint *cons* in *Lib* with an *A*-tuple *args* of arguments, where *A* is the arity of the constraint. The arguments are drawn from the identifiers that appear in *SubModelInstance* as shown in Algorithm 5.6. The *BaseArguments* include the variable and parameter collections whose identifiers appear in *SubModelInstance*, and the same collections restricted to the subsets that are actually used in the constraints of *SubModelInstance*. In addition, the arguments can also be array access expressions composed from the *BaseArguments*, the constant zero, and a special blank symbol. This blank symbol is used as a place-holder for arguments known to be functionally defined by the others. Once all non-blank arguments are selected, the blank symbol is replaced by its corresponding value. For example, in the constraint `maximum(_, xs)` the value of the first argument can be computed from the value of the second and this will be represented by a blank symbol. Note that the value

of the blank symbol must be the same for all sample solutions. If the constraint is not functional, or if the sample solutions disagree on what the value should be — for example, if one sample solution suggests `maximum(10, xs)` but another says `maximum(8, xs)` — they will be discarded in the intersection computation in `process_group`. Note that, since the generated candidate constraint must be the same for all instances, the functionally-defined arguments to this constraint must take the same value *across all instances* to be considered as a candidate, since the constraint cannot be posted to the model without a selected value. This is however not a significant issue, as they are only used when no named parameter is found (i.e., `maximum(a, xs)`, where `a` comes from the model, should be taken over `maximum(8, xs)`).

Finally, the candidate global constraint `cons(args)` is added to the template submodel and the result is evaluated. If the constraint is satisfiable, `cons(args)` is added to the list of candidate global constraints for *SubModelInstance*. If a satisfiable solution for the constraint is not found within a reasonable amount of time we treat it as if it is unsatisfiable. This maintains the correctness of the approach at the cost of completeness.

Let us now illustrate how `generate_candidates` works with the *SubModelInstance* formed by combining the constraint $(g = \min(G)) \wedge (\forall HT : channel)$ of the Progressive Party model (where $\min(G) = 1$) with the variable and parameter declarations in the model, and some data file $DtFile \in DtFiles$. Listing 5.5 presents a MiniZinc instance that corresponds to the selected submodel instance using the data presented in Section 5.2. The *BaseArguments* computed by the procedure for *SubModelInstance* include the variable collections `hostedBy` and `visits`, the variable sub-collections `hostedBy[1,t]` and `visits[1,h,t]`, and the parameter collections `HostBoats`, `GuestCrews`, `p`, `nh`, `ng`, `Time`, and the index `g` itself (with value 1). The *Arguments* computed are the constant 0, the blank symbol, the base arguments, and the array accesses formed by combining an array with a parameter, e.g., `crew[nh]` and `hostedBy[p, ng]`. After considering all constraints in *Lib* with these arguments, the procedure generates 39 candidate constraints including the following:

```
lex2(hostedBy)
alldifferent([hostedBy[g,1], hostedBy[g,2], ..., hostedBy[g,p]])
sub_circuit([hostedBy[g,1], hostedBy[g,2], ..., hostedBy[g,p]])
```

Note that in the first global constraint, the entire `hostedBy` array is used as an argument, while in the second and third global constraints only the subset of the array that participates in the constraints of the submodel – where `g` is fixed – is used as an argument.

Ranking and Filtering

As shown in Algorithm 5.5, once all candidates for a submodel are generated, the method ranks and filters those candidates. Procedure `rank_and_filter`, shown in Algorithm 5.7 achieves this as follows. For each candidate constraint `cons(args)` inferred for a given *SubModelInstance* (and, given the intersection performed by `process_group`, for all previous ones too), `rank_and_filter` measures how closely it matches *SubModelInstance*. To achieve this, it collects a random sample of solutions of `cons(args)`, and computes the fraction of these solutions that are also solutions to *SubModelInstance*. If the constraint is

equivalent to the submodel of *SubModelInstance*, this fraction must be 1; if the constraint is a poor match, the fraction should be close to 0. The procedure filters the candidates by keeping only those constraints whose matching fraction is greater than a given threshold. A threshold of 0.5 has been shown experimentally to be sufficient to eliminate imperfect matches, and we use that value in the implementation. A ranking of the candidates is achieved in `rank_and_filter` by simply sorting them by their *Score*.

Algorithm 5.7 Ranking submodel instances.

```

procedure rank_and_filter(SubModelInstance, Candidates)
  result  $\leftarrow \emptyset$ 
  for each cons(args) in Candidates do
    SolutionsC  $\leftarrow$  random sample of solutions of cons(args)
    SolutionsM  $\leftarrow$  subset of SolutionsC that are also solutions of SubModelInstance
    Score  $\leftarrow |SolutionsM| \div |SolutionsC|$ 
    if Score  $\geq$  equivalenceThreshold then
      result  $\leftarrow$  result  $\cup \langle$  Score, cons(args)  $\rangle$ 
  return sequence of cons(args) from result ranked by Score

```

In some cases a constraint in a model is equivalent to a candidate global constraint only in the context of another constraint. Consider a *SubModelInstance* containing constraints *A* and *B*, and a candidate global constraint *cons*(*args*), where *A* is not equivalent to *cons*(*args*), but the conjunction $A \wedge B$ is equivalent to the conjunction *cons*(*args*) $\wedge B$. We call *B* the *context* in which *A* is equivalent to *cons*(*args*). For example, let *SubModelInstance* have the constraints $(t = \min(T)) \wedge \forall GH : \text{channel} \wedge \forall H : \text{capacity}$ from the Progressive Party. The global constraint `bin_packing_capa(capacity, hostedBy[1..ng,t], crew)` is equivalent to $\forall H : \text{capacity}$, but only in the context of $\forall GH : \text{channel}$. This is due to the fact that the `bin-packing` constraint does not constrain the auxiliary `visits` variables. This connection must be provided by the `channel` constraints. In general, a contextually-equivalent constraint *cons*(*args*) will be implied by *SubModelInstance* but appear weaker than the instance and, thus, will be ranked in a low place. In this case, we try using one of *SubModelInstance*'s constraints as the context constraint *B*, and test via sampling whether *cons*(*args*) $\wedge B$ implies the submodel instance. If this scores well, we say that *A* (obtained as $SubModelInstance \setminus B$) is equivalent to *cons*(*args*) under the context of *B*, and add this to the list of candidates. For the purpose of scoring and filtering, a candidate constraint is thus considered to be a pair of the *cons*(*args*) and its context, which may be empty. This means that for a context-dependent constraint to pass the filtering tests, it must pass with the same context in all instances of the group.

5.4.5 Preprocessing Step to Uncover Hidden Structures

The method proposed in Section 5.4.4 to build the arguments of the candidate global constraints, uses only variable and parameter identifiers already present in the original model. This is not enough to discover constraints that might include dual “hidden” variables, that is, dual viewpoints involving new variables that could be declared and connected to existing model variables via channelling constraints. For example, as seen in Section 2.3,

```

1 predicate channel_2of3(array[int,int] of var int : x,
2                      array[int,int,int] of var int : b) =
3      lb_array(b) in {0,1}
4      /\ ub_array(b) in {0,1}
5      /\ forall (i in index_set_1of3(b),
6                v in index_set_2of3(b),
7                k in index_set_3of3(b))
8          (x[i,k] = v <-> (b[i,v,k]=1));

```

Listing 5.6: Channelling between matrices of integer and binary variables.

it is common in modelling problems to use a set of zero-one variables b_v to represent a single integer decision x , such that $b_v = 1$ if and only if x takes the value v . By extension, a vector of integer decisions x_i can be represented as a matrix of zero-one variables b_{iv} , such that $b_{iv} = 1 \leftrightarrow x_i = v$. Further, a two dimensional matrix of integer decisions x_{ij} can be represented as a three dimensional matrix of zero-one variables b_{ijv} , such that $b_{ijv} = 1 \leftrightarrow x_{ij} = v$. The dimensions used to represent the variables can vary. For example, instead of the third dimension of b representing the selected value, it may be the case the second dimension is used instead. That is, instead of b_{ijk} representing $x_{ij} = k$ it could be $b_{ijk} = 1 \leftrightarrow x_{ik} = j$. This form of channelling can be added to a model using the constraint shown in Listing 5.6. Note that the `IofJ (2of3)` indicates that this constraint is for channelling dimension `I` of a `J`-dimensional array of zero-one variables to a `J-1` dimensional matrix of integer variables. The predicate first posts two constraints restricting the domain of the variables in `b` to be zero-one. This is achieved using the `lb_array` and `ub_array` functions which returns the lowest lower bound, and largest upper bound, respectively, of variables in the multi-dimensional array. Next, a `forall` loop iterates over the index sets of the dimensional array `b`. The function `index_set_IofJ` is used since MiniZinc supports arrays with arbitrary contiguous index sets.

This style of representation is especially common when targeting a mixed-integer programming solver. However, the integer variables in such models may sometimes be omitted, with only the zero-one variables explicitly declared, as their corresponding integer meanings can be computed from solutions. Knowledge of the presence of these variables might allow new global constraints that were previously hidden to be inferred.

This section extends the method to find such “hidden” variables (in particular, implied integer variables), and use them to extend the set of candidate constraints. To achieve this we perform two modifications. The first one is to define a new kind of global constraint that captures the dual representations we are trying to detect. In particular, we consider three global constraints which take a 1-, 2- or 3-dimensional array of zero-one variables as an argument, and hold if a specific dimension potentially represents the values of an integer. For example, the following constraint holds if a 3-dimensional array of variables `b` might represent a 2-dimensional array of integers `x` such that `b[i,v,k]=1 <-> x[i,k]=v`.

```

1 predicate binaries_represent_int_2of3(array[int,int,int] of var int : b) =
2     lb_array(b) in {0,1}
3 /\ ub_array(b) in {0,1}
4 /\ forall (i in index_set_1of3(b),
5           k in index_set_3of3(b))
6     (sum (v in index_set_2of3(b)) (b[i,v,k]) = 1);

```

A second modification adds an initial Globalizer pass to the method aimed at detecting this new kind of constraint. If such a constraint is found, the implied dual variables (i.e, the integer variables) and a channelling constraint linking them to the original ones (i.e., a channelling constraint linking the new integer variables to their original zero-one representatives) are added to the model. This pass allows the method to discover properties of variables as constraints and make them explicit in the model. Once this additional pass has finished, the second pass of Globalizer is launched on the augmented model to find constraints using the introduced variables.

Consider, for example, a model for the Latin Square problem. A Latin Square is an $n \times n$ matrix of integers where each row and column contain permutations of the numbers 1 to n . A solution to this problem for a given a value n , is an $n \times n$ Latin Square. The MiniZinc model for the Latin Square problem presented in Listing 5.7 is a summarised version of the one included in the MiniZinc benchmark set, which uses zero-one variables and linear constraints.

```

1 int: n;
2 set of int: N = 1..n;
3 array [N, N, N] of var 0..1: b;
4 constraint forall (i, j in N) (sum (k in N) (b[i, j, k]) = 1);
5 constraint forall (i, k in N) (sum (j in N) (b[i, j, k]) = 1);
6 constraint forall (j, k in N) (sum (i in N) (b[i, j, k]) = 1);

```

Listing 5.7: Binary model for Latin Squares problem.

Globalizer finds that each dimension of `b` might represent integer variables. That is, it finds the following constraints:

```

1 binaries_represent_int_1of3(b)
2 binaries_represent_int_2of3(b)
3 binaries_represent_int_3of3(b)

```

It then introduces new arrays of auxiliary variables and channelling constraints as follows, with the different versions of the channelling constraint mapping to different dimensions:

```

1 array [index_set_2of3(b),index_set_3of3(b)] of var index_set_1of3(b): b_3a;
2 constraint channel_1of3(b_3a, b);
3 array [index_set_1of3(b),index_set_3of3(b)] of var index_set_2of3(b): b_3b;
4 constraint channel_2of3(b_3b, b);
5 array [index_set_1of3(b),index_set_2of3(b)] of var index_set_3of3(b): b_3c;
6 constraint channel_3of3(b_3c, b);

```

With the introduction of these variables and constraints, Globalizer can now find the alldifferent constraints that correspond to the sets of sum constraints in the original model.

As another example, consider again the Progressive Party problem but, this time, the zero-one integer model given by Smith et al. (1996). A notable difference between this model and the one presented in Listing 5.1 is the absence of the `hostedBy` variables. The discovery of the `alldifferent` and `bin-packing` constraints depended upon these variables being present. With the introduction of this initial pass, Globalizer can now discover these integer variables and, therefore, the global constraints implied by the model. To illustrate this, we have translated the zero-one integer program into MiniZinc and adapted it in such a way that the set of hosts is fixed in advance, as done in Section 6 of Smith et al. (1996). The variable names are also changed to match those in Listing 5.1. This yields the model presented in Listing 5.8.

The first pass of the extended Globalizer can discover that the `visits[g,h,t]` variables represent integer variables $H[g,t]$ such that if `visits[g,h,t]` is 1, then $H[g,t]$ equals h . After introducing these variables, the second pass is able to discover that the first constraint is an `alldifferent` constraint, and the second is a `bin-packing` constraint.

5.5 Implementation

This section provides details regarding the implementation of the globalizing system, called the MiniZinc Globalizer, used in the experiments. In particular, the following subsections discuss several implementation choices taken while implementing the different algorithms, describe the particular composition of the global constraint library used, and provide a quick overview of a web interface that was developed for the MiniZinc Globalizer.

```

1  int : p; int : ng; int : nh;
2  set of int : HostBoats = 1..nh;
3  set of int : GuestCrews = 1..ng;
4  set of int : Time = 1..p;
5
6  array [GuestCrews] of int : crew;
7  array [HostBoats] of int : capacity;
8
9  array [GuestCrews, HostBoats, Time] of var 0..1 : visits;
10
11 constraint forall (g in GuestCrews, t in Time) (
12     (sum(h in HostBoats) (visits[g,h,t])) = 1 );
13     % at_most_one_visit
14
15 constraint forall (h in HostBoats, t in Time) (
16     (sum(g in GuestCrews) (
17         crew[g]*visits[g,h,t])) <= capacity[h]);          % capacity
18
19 constraint forall (h1,h2 in HostBoats, g1,g2 in GuestCrews, t1,t2 in Time
20     where h1 != h2 /\ g1 < g2 /\ t1 != t2) (
21     visits[g1,h1,t1] + visits[g2,h1,t1] +
22     visits[g1,h2,t2] + visits[g2,h2,t2] <= 3 );          % will_meet
23
24 constraint forall (h in HostBoats, g in GuestCrews) (
25     (sum(t in Time) (visits[g,h,t])) <= 1);          % meet_once

```

Listing 5.8: Zero-one integer model for the Progressive Party Problem.

5.5.1 Implementation Choices

The implementation of the `globalize(M, DtFiles, Lib, SubModelSize)` algorithm provided in Section 5.4 assumes the model *M* given as input is a MiniZinc model. As shown in the algorithm, once the submodels are computed by `generate_submodels(Constrs, SubModelSize)`, each of these submodels can be processed independently. Thus, the implementation can process them in parallel to increase efficiency.

As shown in Algorithm 5.6, `generate_candidates(SubModelInstance, Decls, Lib)` first solves *SubModelInstance* to find a random sample of its solutions. To do this, the default implementation uses the standard MiniZinc tool `mzn2fzn` to flatten *SubModelInstance*, and the Gecode constraint solver (Gecode Team, 2006) version 4.4.0 to find 30 random solutions for it (while 1, 30 and 100 were tested in our experiments, 30 was found to have the best accuracy/efficiency ratio and is, thus, used in the default implementation). These are found with a search that selects values in random order and restarts from scratch whenever a solution is found. If the search is not complete within 1 second, *SubModelInstance* is discarded as it may be too expensive to find 30 random solutions for this *SubModelInstance*.

Later in the process, the algorithm needs to determine the satisfiability of the *Instance* after adding candidate global constraints to the *Template*. Since this template has no variables and the added constraints are simply evaluated, no search is required. Such checks are performed often, and it was clear that the expense of flattening the instance and calling a full constraint solver was crippling. To avoid this, a simple evaluator of MiniZinc instances known not to have any variables was implemented. In practice, this optimization is crucial, as the number of evaluations is usually in the hundreds of thousands.

Additionally, choices must be made about the number of samples that should be sought for any global constraint (as opposed to samples for a submodel instance), and the value of *SubModelSize*, i.e., the maximum number of constraints that can appear in the submodels. After considerable experimentation, we settled upon a maximum of two constraints in each submodel, and generating 30 samples for ranking and filtering for the default implementation. The results of our experiments are explored and evaluated in Section 5.6.

Another decision that had to be made was the choice of possible values from which to select for fixing loop variables. We chose $\min(x)$, $\max(x)$, and $\text{mid}(x)$, as special cases in loops tend to be at the boundaries (e.g., the first or last variable in an array often has different constraints on it than the others). The $\min(x)$ and $\max(x)$ cover these edge cases, with $\text{mid}(x)$ selecting a non-boundary case. However, other values could also have of been selected.

5.5.2 Library of Global Constraints

Table 5.1 lists the global constraints appearing in the current implementation of *Lib*, which is used for candidate generation. These are all the global constraints defined in MiniZinc’s standard library (version 1.6) over integer arguments (sets are not handled yet by the prototype implementation), with the addition of the following constraints:

<code>alldifferent</code>	<code>circuit</code>	<code>global_cardinality</code>	<code>nvalue</code>
<code>alldifferent_except_0</code>	<code>count</code>	<code>increasing</code>	<code>sliding_sum</code>
<code>all_equal</code>	<code>cumulative</code>	<code>inverse</code>	<code>sort</code>
<code>atleast</code>	<code>decreasing</code>	<code>lex_less</code>	<code>strict_lex2</code>
<code>atmost</code>	<code>diffn</code>	<code>lex_lesseq</code>	<code>subcircuit</code>
<code>bin_packing</code>	<code>distribute</code>	<code>lex2</code>	<code>unary</code>
<code>bin_packing_capa</code>	<code>element</code>	<code>maximum</code>	<code>value_precede</code>
<code>bin_packing_load</code>	<code>exactly</code>	<code>minimum</code>	
<code>channel</code>	<code>gcc</code>	<code>member</code>	

Table 5.1: Library of global constraints supported by Globalizer.

- `channel(x, a)`: channels an integer variable `x` to an array of zero-one variables `a`.
- `gcc(x, counts)`: a special case of `global_cardinality` where the “cover” argument, which specifies a map from indices to values, is fixed to the identity map.
- `unary(s, d)`: a special case of `cumulative` where the resource capacity and the usage for each task are fixed to 1, implementing a unary resource constraint.

These three global constraints have special versions in some solvers (e.g., Gecode) and are, therefore, worth detecting even if they are special cases of more general constraints. Our implementation is able to evaluate most of the global constraints above using the default decomposition given in the MiniZinc library. However, some decompositions introduce variables that cannot be handled by the simple satisfiability check evaluator (recall that, as explained in Section 5.5.1, the satisfiability check does not perform search). Thus, stand-alone constraint checkers were implemented in these cases to evaluate these globals.

As mentioned in Section 5.4.4, each global constraint is annotated with conditions for its use in order to prevent the constraint from being considered as a candidate when it is trivially true or otherwise useless. For example, the `alldifferent` global constraint specifies that its argument must be an array of variables with arity greater than one since, otherwise, the constraint is trivially true or nonsensical. As another example, the `sliding_sum(l, u, n, x)` global constraint specifies that every `n`-length subsequence of `x` must sum to a value between `l` and `u`. When generating candidates, we ensure that $l < u$ and $1 < n < \text{length}(x)$. These two conditions ensure that the parameters make sense, thus avoiding the generation of meaningless candidates.

5.5.3 Displaying Candidates

The MiniZinc Globalizer was implemented as an asynchronous web server that queues requests made by clients and can execute several requests in parallel. Requests can be cancelled by the user and are automatically cancelled when the session is terminated, e.g., when a user closes the browser window. The online service is no longer available and an off-line version that integrates with the MiniZinc IDE is currently in development.

Figure 5.2 shows a screen shot of the web interface which is similar to the interface provided by the MiniZinc IDE. Users can enter their model and instance data through an

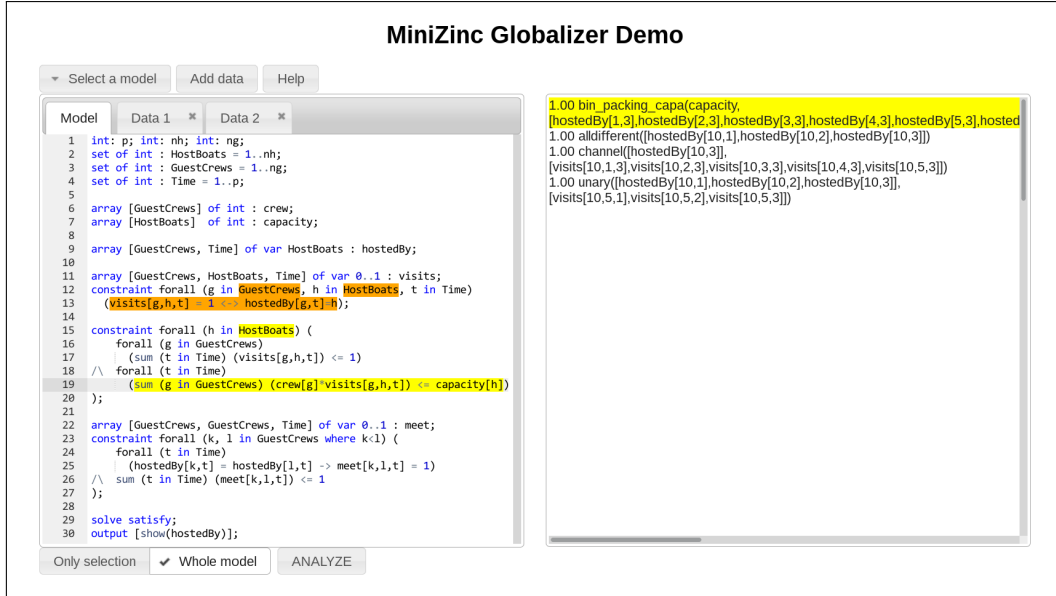


Figure 5.2: The web interface to the MiniZinc Globalizer.

embedded editor (seen on the left). Clicking **ANALYZE** launches the request, initiating a progress bar that attempts to estimate the percentage of work left and provides the user periodic progress updates. When the analysis finishes, the right hand side of the window displays the results in ranking order (indicated by the number preceding each entry, where confidence is given as a value in the interval $[0,1]$). Clicking on any candidate constraint highlights in yellow the part of the original model that the constraint could replace, and highlights in orange the candidate’s context, if any. The interface allows the user to select parts of the model and restrict the analysis to the selected parts by clicking “Only selection” in the lower left corner of the window. The analysis will then only use the selected constraints. This is useful when the analysis is taking a long time, or the user wants to focus on a particular part of their model.

5.6 Experimental Evaluation

This section reports on a set of experiments designed to evaluate the accuracy and practicality of the default implementation of the MiniZinc Globalizer. In addition, it reports on another set of experiments that shows the reasons behind the choices made for our default implementation. Note that the experiments attempt to find globals using each model in its entirety, without using the “Only selection” feature provided by the web interface which can speed up the search in many cases.

5.6.1 Evaluating the Default Configuration

Let us start by discussing the results of evaluating the accuracy and practicality of the default configuration of the implementation (detailed in Section 5.5.1). The results are shown in Table 5.2 where, for each problem model, the table displays the name of the problem (**problem**), the time in seconds to run the MiniZinc Globalizer (**time**), the

number of global constraint candidates found with a score > 0.5 ($|\mathbf{cs}|$), the number of submodel instance groups obtained ($|\mathbf{sms}|$), the number of calls to Gecode to obtain sample solutions of either submodel instances or global constraint candidates (**calls**), the number of satisfiability tests performed (**evals**), and some of the global constraints proposed as candidates with a score of 1 (**top candidates**), where high quality candidates appear in bold. Note that the output has been manually simplified, with some duplicate constraints removed where the system was unable to distinguish between two parameters that appear to be different, but actually refer to the same value. Two data files are used for each model and the default parameters for number of samples to generate (30), and number of constraints per submodel (2) are used. The models used in the experiments are reproduced in Appendix B. The set of problems used in the table is as follows.

Cars: a version of the car sequencing problem of CSPLib 001, as implemented in the MiniZinc distribution. It uses simple arithmetic and counting constraints to express the capacity and sequence restrictions of the problem. Globalizer finds three constraints: `sliding_sum`, `count` and `gcc`. The `gcc` subsumes the `count` constraint.

Jobshop: a simple job-shop scheduling problem taken from the MiniZinc distribution. It implements the non-overlapping of two tasks on a unary resource using simple reified constraints. Globalization finds a single constraint: the `unary` scheduling constraint.

Party, Party-CSP and Party-LP: the running example from Listing 5.1. Globalizer finds 6 different constraints. The most important of which were discussed earlier: the `bin_packing`, `alldifferent`, and `channel` global constraints. It also finds a `unary` scheduling constraint. **Party-CSP** is a slight variant of the **Party** model. The main difference is that the constraints are not grouped together under conjunctions (\wedge) under `forall` loops. It also includes the `alldifferent` constraint on the `hostedBy` variables. The results presented in the table show that this has little impact on the amount of time it takes to discover globals. However, we see that some constraints are not found, with only 3 constraints being found here, compared to the 6 of the first model. **Party-LP** is the integer model presented in Listing 5.8. Globalizer first detects and introduces integer variables that channel to `visits` named `visits_3b`. These correspond to the `hostedBy` variables from Listing 5.1. It then successfully detects the `alldifferent` and `bin_packing_capa` constraints on the `visits_3b` variables.

Packing: a packing problem that aims to pack n squares into a rectangle. The source code was taken from the MiniZinc distribution. Globalizer finds `diffn` constraints that express the non-overlapping of the squares.

Schedule: a contrived scheduling example from Bessiere et al. (2007). The schedule is constrained in such a way that one task needs to start at every even time point, which implies a `gcc` (global-cardinality) constraint with argument $[1, 0, 1, 0, \dots]$. The approach can find this constraint, as long as all data files provided have the same schedule length, as otherwise the arguments differ in length between instances and are thus discarded. The generalization of such sequences is left to future work.

Sudoku LP and Sudoku CSP: two different models for the Sudoku puzzle where values in each of the nine rows, the nine columns and the nine 3×3 blocks must be distinct. The first, **Sudoku LP**, uses a zero-one integer program formulation, and Globalizer finds 17 constraints, these are some of the `alldifferent` constraints on the rows and columns. It does not however detect the constraints on the blocks since the system does not attempt to generate complex array slices. This issue is discussed in more detail in Section 5.7.2. In addition to the `alldifferent` constraints, some `global-cardinality` constraints are also detected which are equivalent to the `alldifferent` in stating that the values $1, 2, \dots, 9$ can only occur in each row or column a single time. The second model, **Sudoku CSP**, represents the entire problem with a single `forall` loop over every pair of variables in the matrix. A complicated `if` statement then decides whether a `not-equal` constraint should be posted between each pair or not. Here, loop unrolling is not strong enough to generate candidates that correspond to individual rows, columns, or blocks. As a result, no replacement global constraints can be found. This motivates somewhat the use of a syntactic matching approach in conjunction with Globalizer, as these constraints may have been easier to detect in this case.

Warehouses: the warehouse allocation problem of CSPLib 034, as distributed with the MiniZinc distribution. Globalizer finds that a loop containing counting constraints can be aggregated into one `global-cardinality` constraint.

The experimental results indicate that the default implementation is quite accurate, as in most cases (e.g., **Cars**, **Party**, **Packing**) Globalizer has been able to find the global constraints that an expert modeller would have used. In the other cases (e.g., **Sudoku CSP**), it mainly found constraints with low rank or it simply found nothing useful.

The results show that the approach can be time consuming. The time to analyse a reasonably complex model with 3-4 data sets is in the range of several minutes up to an hour, depending mainly on the number of candidates that need to be checked for satisfiability, a number that grows considerably with the number of possible arguments. However, this is not unacceptably slow, as it should not be necessary to use Globalizer often and, if it can find global constraints for the user's model, it can potentially speed up the solving of all resulting instances, which more than makes up for the initial investment. Further, the current prototype is not optimized for performance. There is still great potential for improving performance by, for example, avoiding unnecessary candidate checks and parameter instantiations which, as indicated, make up the bulk of the run time.

The number of generated submodel instance groups (`|sms|`) is relatively small (usually less than 100), which means that the problem splitting algorithm achieves a good level of pruning. Generating only a small number of groups and top scoring constraints is important, since the results are meant to be presented to a human user.

Looking at the number of satisfiability tests, which can reach hundreds of thousands, it becomes clear that each check needs to be efficient. This justifies the introduction of a dedicated constraint evaluator as discussed in Section 5.5.1.

problem	time	cs	sms	calls	evals	top candidates
Cars	37.3	3	35	848	70173	<code>gcc(step_class,cars_in_class)</code> <code>count(step_class,c,cars_in_class[c])</code> <code>sliding_sum(0,</code> <code>option_max_per_block[p],</code> <code>option_block_size[p],</code> <code>step_option_use[1..10,*])</code>
Jobshop	534.3	1	99	13704	1483093	<code>unary(s[1..n,*], d[1..n,*])</code>
Party	809.2	6	280	6438	308482	<code>bin_packing_capa(capacity,</code> <code>hostedBy[i..n,*],</code> <code>crew)</code> <code>alldifferent(hostedBy[*,1..4])</code> <code>channel(hostedBy[*,*], visits[*,1..4,*])</code> <code>unary(hostedBy[*,1..4], visits[*,*,1..4])</code>
Party-CSP	850.3	3	258	6799	485166	<code>bin_packing_capa(capacity,</code> <code>hostedBy[i..n,*],</code> <code>crew)</code> <code>alldifferent(hostedBy[*,1..4])</code> <code>channel(hostedBy[*,*], visits[*,1..4,*])</code>
Party-LP	128.7	4	79	1322	55084	<code>alldifferent(visits_3b[g,*])</code> <code>bin_packing_capa(capacity, visits_3b[*,t], crew)</code> <code>binaries_represent_int_3B(visits)</code>
Packing	166.1	2	163	4895	426052	<code>diffn(x,y,pack_s,pack_s)</code> <code>diffn(y,x,pack_s,pack_s)</code>
Schedule	44.3	1	42	1241	99285	<code>gcc(x,[1,0,1,0,1,0,1])</code>
Sudoku LP	21339.9	17	580	15762	974174	<code>alldifferent(p[*,1..9])</code> <code>gcc(p[*,1..9], [1,1,1,1,1,1,1,1,1])</code> <code>alldifferent(p[1..9,*])</code> <code>gcc(p[1..9,*], [1,1,1,1,1,1,1,1,1])</code>
Sudoku CSP	4230.5	0	61	5116	364467	-
Warehouses	68.7	1	97	1793	227023	<code>gcc(supplier,use)</code>

Table 5.2: Globalizer experiment (3 datafiles, 30 samples, 2 constraints).

5.6.2 Evaluating Alternative Configurations

Let us now evaluate configuration choices other than those taken in the default implementation, in particular, the effect of varying the number of data files per problem used to instantiate submodels (3 in the default configuration), the number of sample solutions used to build the templates and filter the global constraint candidates (30 in the default configuration), and the size of submodels (2 constraints in the default configuration), on the accuracy and practicality of the resulting prototype. Note that while currently there are few enough possible parameter configurations to be explored that this kind of analysis can be done by hand, in future, given the introduction of more heuristic components to the Globalizer, it may be useful to apply machine learning approaches to tune parameters. Table 5.3 shows the results of this evaluation, which needs to be considered together with those presented in Table 5.2. Each cell in the table shows the running time (**time**), the number of global constraints found (**|cs|**), the number of submodel instance groups generated (**|sms|**), the number of calls to the constraint solver (**calls**), and the number of satisfiability checks performed (**evals**). Figure 5.3 shows plots demonstrating in graphical form several interesting results from these experiments. The plotted values are ratios, with the first configuration being the baseline.

The results shown in the first row of Table 5.3 (together with those in Table 5.2) indicate that increasing the number of data-files often reduces the number of global constraint candidates generated by the method. For example, analysing the **packing** problem with

	time	cs	sms	calls	evals	time	cs	sms	calls	evals
problem	1 Datafile					2 Datafiles				
Cars	26.9	4	35	543	65923	33.5	3	35	748	72292
Jobshop	262.4	2	99	6696	891532	268.6	2	99	6877	883609
Party	831.9	6	280	3731	313166	817.7	5	280	5082	311055
Party-CSP	993.8	3	258	4394	495079	893.8	3	258	5526	484318
Party-LP	77.3	4	79	637	38180	103.4	4	79	973	46014
Packing	138.8	20	163	4337	452842	153.7	2	163	4835	457195
Schedule	34.6	1	42	914	99327	41.4	1	42	1107	102138
Sudoku LP	8655.7	33	580	7170	514396	14979.0	2	580	114515	742875
Sudoku CSP	1483.0	0	61	2736	238385	2854.3	0	61	3926	301604
Warehouses	56.7	1	97	1340	209726	68.5	1	97	1774	225933
	1 Sample					100 Samples				
Cars	45.4	3	35	1402	32176	53.5	3	35	825	157532
Jobshop	2517.4	62	99	76720	526517	680.8	1	99	12443	3787542
Party	645.7	30	280	15366	176552	1995.2	5	280	6296	722161
Party-CSP	566.6	63	258	14426	253512	2028.0	3	258	6539	1017368
Party-LP	61.0	5	79	1711	8718	305.3	4	79	1317	168493
Packing	287.6	24	163	10787	135731	237.4	2	163	4685	1095227
Schedule	68.4	9	42	2518	38107	64.8	1	42	1211	242938
Sudoku LP	1871.8	26	580	16850	88644	67712.4	14	580	15557	3057030
Sudoku CSP	421.3	0	61	6194	84632	13534.6	0	61	5116	1044746
Warehouses	139.6	23	97	4761	120312	90.1	1	97	1622	443992
	1 Constraint					3 Constraints				
Cars	31.5	3	27	712	56472	38.0	3	36	860	71291
Jobshop	400.6	1	60	10075	1114608	559.2	1	113	14340	1560720
Party	505.5	5	140	3879	194213	928.1	5	352	7366	336937
Party-CSP	546.5	3	130	4239	298793	942.8	3	327	7636	524926
Party-LP	110.5	4	52	1070	47075	133.0	4	87	1379	57623
Packing	79.8	2	78	2362	195102	259.6	2	243	7516	662124
Schedule	31.6	1	30	888	68360	44.9	1	43	1262	101126
Sudoku LP	18308.6	18	216	10700	742623	23984.7	21	1060	21010	1220339
Sudoku CSP	4125.2	0	40	3420	243729	4229.3	0	61	5116	365088
Warehouses	26.6	1	45	685	76511	108.8	1	130	2896	369605

Table 5.3: Comparison of different configurations.

a single data file results in 20 candidate global constraints with a score of 1. Adding the additional data files increases the discriminative power of the system by reducing the candidates with a score of 1 to the two shown in Table 5.2. Similarly, analysing **Sudoku-LP** with only one data file results in 33 candidate global constraints with a score of 1, as opposed to two given two data files. However, for some problems such as **Schedule** and **Warehouses**, a single data file was enough to narrow the candidates down to a single constraint with a score of 1. The results are represented graphically in Figure 5.3. Different colours represent different models.

Figure 5.3a (which includes the results from Table 5.2) shows graphically that increasing the number of data files tends to increase also the running time, although in some cases it actually reduces it. The return for this (sometimes considerable) overhead is greater assurance of the validity of the detected global constraints. Note in particular the **packing** problem, where considering 1 data file causes many spurious constraints to be detected, as they relate only to that specific data file. When the other data files are added these constraints are eliminated, leaving only those which are found in all the data files.

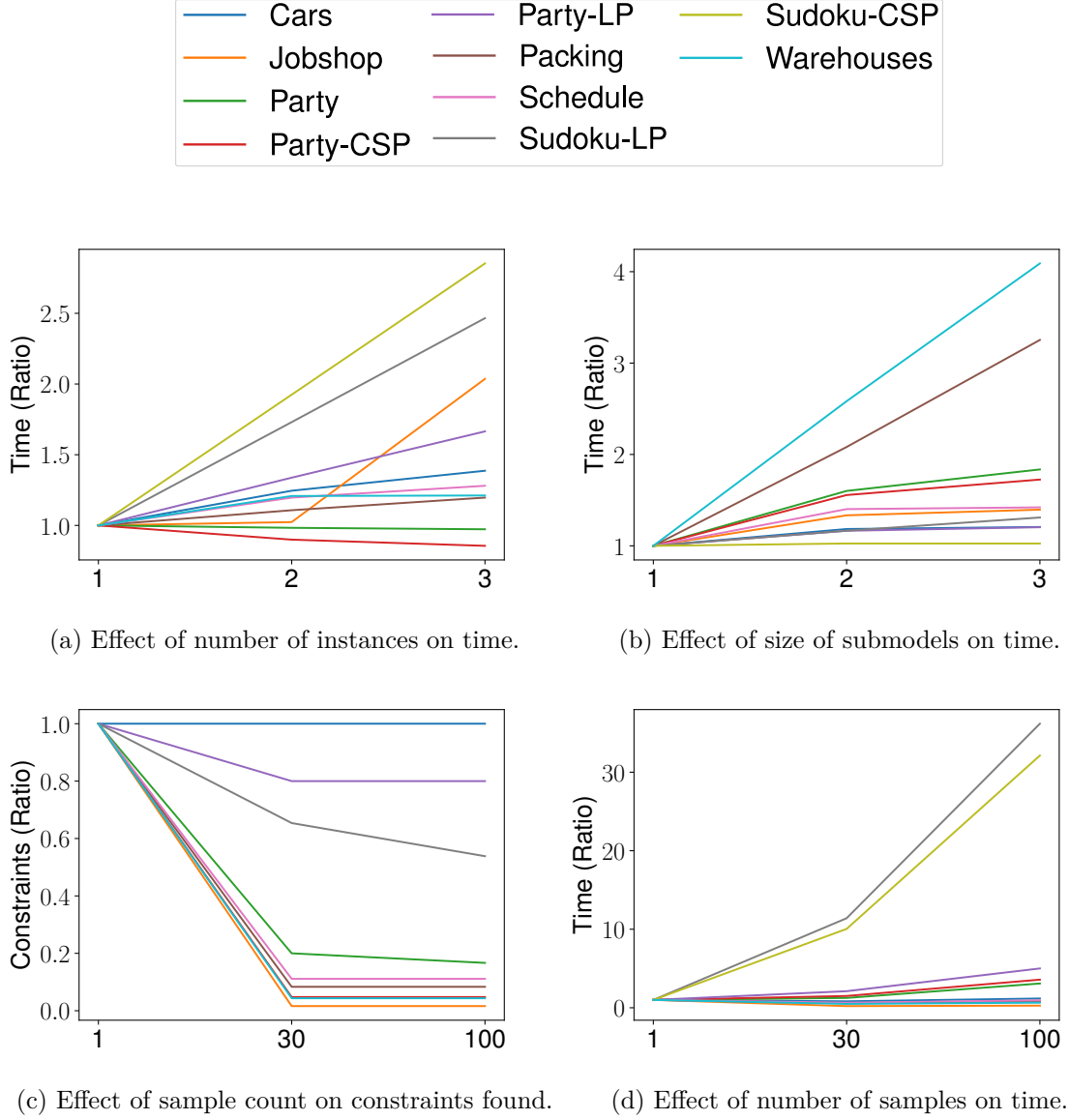


Figure 5.3: Experiment looking at different configurations.

Figure 5.3b shows graphically, that increasing the number of constraints in a sub-model (increasing the size) has the expected effect of increasing the runtime. While for many models the increase is sub-linear, in some cases it can really blow up. For example, the *Warehouses* model takes just over four times longer when submodels can contain up to three constraints. This is why the default configuration uses 2 constraints.

As shown in the second row of Table 5.3, and graphically shown in Figure 5.3c, increasing the number of samples tends to reduce the number of spurious global constraint candidates found, by eliminating those that meet the conditions simply by chance. Note that the number of constraints found under 30 samples (shown in Table 5.2) and 100 samples are quite similar, which is why we chose 30 for our default implementation. An interesting, non-intuitive result can be found in the progressive party model. Here we find a **cumulative** constraint with arguments: `hostedBy[_ , t]`, `visits[_ , h, t]`, `crew`, and `capacity[h]`, which encodes:

```
sum (g in GuestCrews) (crew[g]*visits[g,h,t]) <= capacity[h]
```

This plays a similar role to the **bin-packing** constraint, but the use of **cumulative** requires a different perspective: instead of packing objects (guest crews) into capacitated bins (host boats), we are scheduling tasks (guest crews) such that the resource (host boat capacity) is not exhausted, and the duration of each “task” is 1 if the crew visits the host in that time slot and zero otherwise. The **cumulative** constraint is eliminated in the 100 sample run because an edge case (all tasks with zero duration) is not permitted by the definition of **cumulative**, and the edge case is only found with the larger sample size. This demonstrates Globalizer’s ability to present alternative viewpoints to the modeller, in this case presenting this subproblem as a scheduling task rather than as a bin-packing task.

Figure 5.3d shows that increasing the number of samples typically results a small increase or decrease in running time, which seems a reasonable price to pay for the increased accuracy. Note however, the two outliers of **Sudoku-LP** and **Sudoku-CSP** where we can see a huge increase in execution time. Further analysis showed that, compared to the other benchmark problems, the submodels produced for these models involve a huge number of constraints, and sample generation required exploring very deep search trees. This indicates that the constraints are quite weak. The combination of weak propagation and a huge number of variables and constraints to in a very deep search leads to the comparatively poor performance.

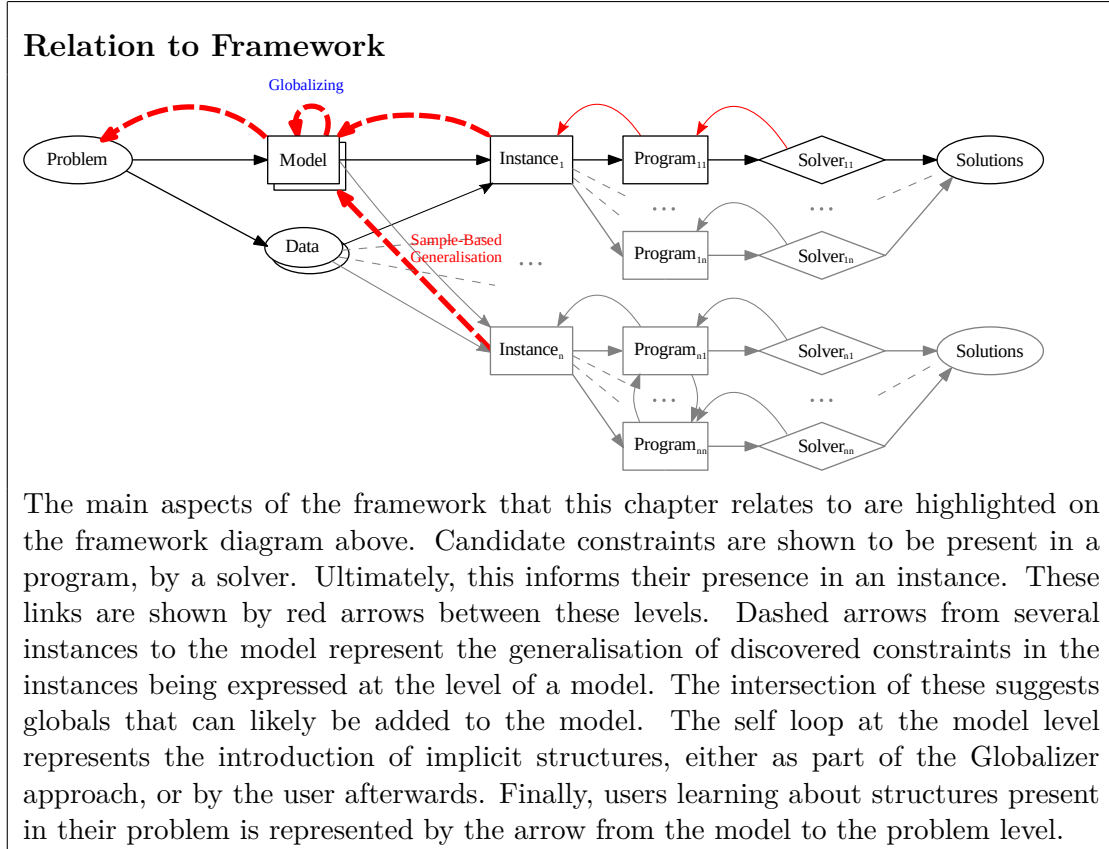
5.7 Conclusion

This chapter proposes a method for *globalizing* constraint models, that is, for inferring global constraints that can replace parts of a given constraint model. This helps users improve their models, since global constraints capture the inherent structure of a model and can, thus, help obtain a better translation to the underlying solving technology and faster solving using specialised algorithms.

The inference process is based on splitting a model into submodels that correspond to subsets of its constraints, potentially unrolling loops, and instantiating each of the resulting submodels with different data sets into a group of submodel instances. From these groups of instances, candidate constraints are generated by sampling the solution space of both the group and the candidate constraints. The candidates are ranked and filtered based on how well their search spaces match.

In addition, the same approach can be applied as an initial pass that focusses on finding alternative viewpoints in the form of binary variables that may represent integer variables. These viewpoints can then be added to the model so that in the second pass with the full set of constraints, Globalizer may be able to find constraints on the variables of these alternative viewpoints.

Experimental evidence was presented showing that the method is both practical and accurate. Experiments were also presented that explore different configurations of the approach. The implementation, the MiniZinc Globalizer, was previously available as a web-based tool. A new version, which integrates with the MiniZinc IDE is planned.



5.7.1 Limitations

Currently, globalizing a model using Globalizer can be a slow process. While we believe that this is a reasonable cost to pay considering how the improvement of a model can speed up the solving of many resulting instances, it is still something that could be improved. The implementation of heuristics that would prioritise the order in which submodels are explored, could allow global constraint to be discovered earlier. In a similar vein, the system currently may test whether a constraint occurs in a subset of a set of constraints that have already been explored. It is unclear how much of a performance boost the pruning of these redundant submodels would provide.

5.7.2 Further Research

While the system already provides useful results, there are some improvements that would be interesting to explore.

Stronger ranking and filtering. One, is the incorporation of ranking techniques from Constraint Seeker, such as using known implications between constraints to eliminate more imperfect candidates. Model Seeker could also be used to generate more complex candidates for each submodel. The confidence in the suggested constraints may also be improved by using theorem proving or other techniques to prove equivalence in cases where it is possible.

MiniZinc IDE integration. A valuable contribution would be the integration of the system with the MiniZinc IDE that lets users refactor models automatically using the suggestions generated by Globalizer.

Array slicing. A natural application of the tool is in teaching constraint programming. In a recent assignment given as part of a constraint programming course at Monash University, students were required to write a MiniZinc model to solve a vehicle routing problem. The problem involved determining a tour to deliver goods to customers over several days using several trucks, where each customer has some fixed demand and each truck some fixed capacity. A constraint of the problem is that the trucks cannot be overloaded. Several students modelled this constraint as follows:

```

1  % Parameters
2  array[1..NumCustomers] of int: Demand;
3  array[1..NumVehicles] of int: Capacity;
4  int : TourLength = NumCustomers + 2*NumVehicles
5  % Assignment of customers / depots to vehicles
6  array[1..TourLength] of var 1..NumVehicles: Assignments;
7  ...
8  % Demand/capacity constraint
9  constraint forall(v in Vehicles) (
10     sum(i in 1..NumCusts)
11         (Demand[i]*(Assignments[i]==v)) <= Capacity[v]);

```

This conjunction of constraints can be replaced by a **bin-packing** constraint, where the trucks are the bins, and the customers are the objects to be placed in the bins. Note that the tour length has more entries than the number of customers, because it also includes the start and end visits to the depot. The replacement constraint could be written as:

```
bin_packing_capa(Capacity, [Assignments[i] | i in 1..NumCustomers], Demand);
```

However, this constraint cannot be found by Globalizer, as it does not construct “array slices”, like the second argument above, as potential arguments. A slice of the array is necessary because the `Demand` array is smaller than the `Assignments` array, as the `Assignments` array also contains the visits to the depots, and for the **bin-packing** we want to consider only the visits to the customers. We also saw this earlier in Table 5.2 where for the **Sudoku LP** model the **alldifferent** constraints on sub-squares of the matrix could not be found.

One approach would be to modify Globalizer to generate every possible array slice as a possible argument. However, this might well be too expensive. Instead, we can modify Globalizer to analyse syntactically the array access expressions in the constraint to determine which elements of the array are used, and use this subset as a potential argument. We expect this to work in most cases.

In the example above, we can see that the `Assignments` array is only used with `i` as an argument, and `i` is declared to be drawn from the set `1..NumCusts`. Via this simple syntactic analysis, we can see that the constraint acts only on the subset of the array given by `[Assignments[i] | i in 1..NumCustomers]`. Therefore, we can consider this slice of the `Assignments` array as a potential argument.

Chapter 6

Structure Guided Fault Diagnosis

6.1 Introduction

As discussed in previous chapters, writing correct and efficient models for problems requires a lot of expertise. In Chapter 5 a method was presented that can help users improve a given, correct model of a problem. Unfortunately, real-world problems typically exhibit a level of complexity that makes it difficult to create a correct model in the first place, with a first attempt at modelling a problem often resulting in an incorrect model. Models may be incorrect in many different ways. This chapter focusses on the case of over-constrained models, where the programs are unexpectedly *unsatisfiable*.

When faced with unsatisfiability, users have few tools available to help them with debugging. The typical strategy for finding the cause of unsatisfiability consists of manually activating and deactivating constraints in the model until a satisfiable subset is found, indicating the problem lies in (some of) the deactivated constraints. This approach is tedious and often impractical since the fault may involve a non-trivial combination of groups of constraints and instance data. Thus, automatic approaches are required to make the task feasible. This chapter aims to answer the third, and final sub-goal presented in Chapter 1:

Can the explicit structure of a model be used to help identify parts of the model that are incorrectly formulated?

To answer this question the chapter presents a method that uses the explicit structure to guide the search for conjunctions of constraints that lead to unsatisfiability.

Motivation

Most automatic techniques for debugging unsatisfiable constraint problem specifications focus on finding *Minimal Unsatisfiable Subsets* (MUSes) (Bailey et al., 2005; Junker, 2001; Liffiton et al., 2013) in constraint programs. A MUS is a set of unsatisfiable constraints that becomes satisfiable if any constraint in the set is removed. MUSes are therefore useful in identifying the possible *sources* of unsatisfiability in a constraint program. This is why the approach presented in this chapter builds upon MUS detection techniques.

Unfortunately, MUS detection techniques do not scale well for constraint programs in general, as these programs typically contain hundreds or thousands of constraints, even

for relatively simple models. To counter this, some debugging techniques focus on specific kinds of constraint programs, such as numeric CSPs (Gasca et al., 2007) or linear programs (Gleeson et al., 1990; Loon, 1981), as the particular properties of these constraints can be used to improve scalability. Other debugging techniques focus on diagnosing unsatisfiability during search (Jussien et al., 2001; O’Callaghan et al., 2005; Ouis et al., 2003), which considerably reduces the scalability problem and can be useful for interactive solving, where users have some input into how the search should proceed when it encounters difficulties. Instead, we must focus on constraint-system agnostic diagnosis algorithms, as our aim is to be able to debug any model, without regard for the type of constraints or particular solving technology used. This will make the scalability of the approach challenging.

Further, the MUSes detected for a constraint program might not be meaningful for users. This is because, as discussed in Chapter 2, much of the explicit model structure is lost during compilation to a program. As a result, a MUS defined in terms of the program might bear little resemblance to the model the user actually designed. This is why most debugging techniques work at the program level, rather than at the model level, making the usefulness of the approach challenging.

Contribution

This chapter builds upon the techniques presented in Chapter 4 to present an automatic approach to the debugging of unsatisfiable constraint problem models that is both scalable and useful. To achieve scalability, the approach extends the concept of variable paths to preserve the constraint structure present in a model and **uses this explicit model structure preserved during compilation to speed up a MUS detection algorithm**. To achieve usefulness, the approach presents the resulting MUSes to the user in a meaningful way by **automatically linking the program constraints in the MUS to the model constraints**. Further, using a generalisation approach similar to that presented in Chapter 5, MUSes found across a *set* of instances can be *generalised*, that is, expressed in terms of the high-level (parametric) model. These **generalised MUSes can help the user to distinguish between genuine modelling bugs from the unsatisfiability that arises from instance data**.

The approach, designed and implemented for the MiniZinc modelling language, can utilise any constraint-system agnostic MUS enumeration algorithm. The proof of concept implementation, experimentally evaluated in Section 6.5, makes use of the MARCO algorithm from Liffiton et al. (2015), and shows that using the structure available in a MiniZinc model can be both useful and efficient, speeding up the search for MUSes.

Structure of the Chapter

Section 6.2 introduces a simple model that will be used as a running example throughout the chapter. Section 6.3 presents an overview of MUS detection algorithms and discusses related work. Section 6.4 introduces the new approach for guiding MUS detection using the explicit structure present in a user’s model. It also discusses how this can be

```

1  int: n;
2  array[1..n, 1..n] of var 1..n: X;
3
4  % Problem constraints
5  constraint forall (i in 1..n)
6      (all_different(row(X, i)));           % rows
7  constraint forall (j in 1..n)
8      (all_different(col(X, j)));           % cols
9
10 % Conflicting symmetry breaking constraints
11 constraint forall (r in 1..n-1)
12     (lex_less(row(X, r), row(X, r+1))); % row ordering
13 constraint forall (c in 1..n-1)
14     (lex_greater(col(X, c), col(X, c+1))); % col ordering
15
16 solve satisfy;

```

Listing 6.1: Latin Squares model with conflicting symmetry breaking constraints.

implemented as part of the simplified MiniZinc interpreter presented in Chapter 2. Section 6.5 presents an experimental evaluation of the prototype on a set of faulty models showing promising speed-ups. Section 6.6 shows how the additional explicit structure can be used to present more meaningful diagnoses to the user, and Section 6.7 discusses how the approach presented can help generalise the results found across several instances to the model-level. Finally, Section 6.8 presents the conclusions to this chapter.

6.2 The Latin Squares Problem

This section presents the model that will be used as a running example to illustrate the techniques presented in this chapter. Recall that a Latin Square is an $n \times n$ matrix of integers where each row and column contain permutations of the numbers 1 to n . Given a value for n , the aim of the problem is to find an $n \times n$ Latin Square. Listing 6.1 presents a faulty MiniZinc model for this problem. Line 1 declares an integer parameter ‘ n ’, to represent the dimension of the matrix. Line 2 creates the $n \times n$ matrix, named `x`, of integer variables that must take values in the range $1..n$. Lines 5 and 6 introduce the first of the main problem constraints, namely, the constraints on the rows of the matrix. Rather than declaring a set of individual `alldifferent` constraints for each row in a specific instance, the modeller has used a `forall` to parametrically group the `alldifferent` constraints, and has used the helper function `row` to select each specific row. Lines 7 and 8 do the same for the columns of the matrix.

The Latin Square Problem has many symmetric solutions (Mears et al., 2008), including symmetries along the horizontal and vertical axes. The modeller has added symmetry-breaking constraints to help the solver avoid the enumeration of these symmetric solutions. These constraints are added on lines 11-14 and introduce a set of `lex-less` and `lex-greater` constraints on pairs of rows and pairs of columns. The `lex_less` constraints on lines 11 and 12 enforce an ordering on the values in the columns, which forces pairs of consecutive rows to be lexicographically ordered from the first to last element. This means that the first element of the first row must be less than or equal to that of the second row, and

```

1 % lex_greater.mzn
2 include "lex_less.mzn";
3 predicate lex_greater(array[int] of var int: x,
4                       array[int] of var int: y) = lex_less(y, x);

```

Listing 6.2: `lex_greater` is rewritten in terms of `lex_less`.

```

1 % lex_less.mzn
2 predicate lex_less(array[int] of var int: x,
3                   array[int] of var int: y) = lex_less_int(x, y);
4
5 predicate lex_less_int(array[int] of var int: x, array[int] of var int: y)
6 = let { int: lx = min(index_set(x)), int: ux = max(index_set(x)),
7         int: ly = min(index_set(y)), int: uy = max(index_set(y)),
8         int: size = max(ux - lx, uy - ly),
9         array[0..size+1] of var bool: b
10     } in
11     b[0]
12     /\
13     forall(i in 0..size) (
14         b[i] = ( x[lx + i] <= y[ly + i]
15                 /\
16                 (x[lx + i] < y[ly + i] \/ b[i+1]) )
17     )
18     /\
19     b[size + 1] = (ux - lx < uy - ly);

```

Listing 6.3: Decomposition of `lex_less` into simple relational constraints.

if all but the final elements are equal these last element must be ordered. For example, the arrays $a = [1, 2, 2]$ and $b = [1, 2, 3]$ satisfy the constraint `lex_less(a, b)` as the final element in a is strictly less than that in b . When combined with the `alldifferent` constraints, they ensure the first column of x is sorted from the lowest to the highest value. This breaks the symmetry on the horizontal axis. Lines 13 and 14 then introduce a set of `lex-greater` constraints which, when combined with the `alldifferent` constraints, similarly restrict the first row to be sorted, this time from highest to lowest. This breaks the symmetry on the vertical axis. Unfortunately, these symmetry-breaking constraints overlap in an undesirable way, as they constrain, for example, the first element of the matrix to be both the smallest and largest value, thus leaving the domain of this variable empty. It is not immediately clear just looking at the model that these symmetry breaking constraints are in conflict. In fact, adding symmetry breaking constraints that successfully eliminate different symmetries without eliminating non-symmetric solutions is a notoriously difficult task and the source of many modelling errors.

For the purposes of demonstrating the strengths of the technique without using an overly complex model, we will assume a compilation that decomposes the `all_different` and `lex` constraints. As explored in previous chapters, the decomposition of `all_different` is quite straight forward, introducing a clique of `not-equals` constraints, one for each pair of variables passed as an argument. The standard library decompositions for the `lex_greater` and `lex_less` constraints are presented in Listing 6.2 and Listing 6.3, respectively. In the former, a `lex_greater` constraint is simply rewritten as a `lex_less` with its arguments

```

array [1..2] of int: i10 = [1,-1];
var 1..3: i0; var 1..3: i1; var 1..3: i2;
var 1..3: i3; var 1..3: i4; var 1..3: i5;
var 1..3: i6; var 1..3: i7; var 1..3: i8;
array [1..9] of var int: X = [i0,i1,i2,i3,i4,i5,i6,i7,i8];
var bool: b18; var bool: b19; var bool: b21; var bool: b24; var bool: b25;
var bool: b26; var bool: b27; var bool: b29; var bool: b32; var bool: b33;
var bool: b35; var bool: b38; var bool: b39; var bool: b40; var bool: b41;
var bool: b43; var bool: b46; var bool: b47; var bool: b49; var bool: b52;
var bool: b53; var bool: b54; var bool: b55; var bool: b57; var bool: b60;
var bool: b61; var bool: b63; var bool: b66; var bool: b67; var bool: b68;
var bool: b69; var bool: b71;
int_lin_ne(i10,[i0,i1],0);
int_lin_ne(i10,[i0,i2],0);
int_lin_ne(i10,[i1,i2],0);
int_lin_ne(i10,[i3,i4],0);
int_lin_ne(i10,[i3,i5],0);
int_lin_ne(i10,[i4,i5],0);
int_lin_ne(i10,[i6,i7],0);
int_lin_ne(i10,[i6,i8],0);
int_lin_ne(i10,[i7,i8],0);
int_lin_ne(i10,[i0,i3],0);
int_lin_ne(i10,[i0,i6],0);
int_lin_ne(i10,[i3,i6],0);
int_lin_ne(i10,[i1,i4],0);
int_lin_ne(i10,[i1,i7],0);
int_lin_ne(i10,[i4,i7],0);
int_lin_ne(i10,[i2,i5],0);
int_lin_ne(i10,[i2,i8],0);
int_lin_ne(i10,[i5,i8],0);
array_bool_and([b25,b26],b18);
array_bool_and([b27,b29],b19);
int_lin_le_reif(i10,[i0,i3],-1,b21);
array_bool_or([b18,b21],true);
int_lin_le(i10,[i0,i3],0);
int_lin_le_reif(i10,[i1,i4],-1,b24);
array_bool_or([b19,b24],b25);
int_lin_le_reif(i10,[i1,i4],0,b26);
int_lin_le_reif(i10,[i2,i5],-1,b27);
int_lin_le_reif(i10,[i2,i5],0,b29);
array_bool_and([b39,b40],b32);
array_bool_and([b41,b43],b33);
int_lin_le_reif(i10,[i3,i6],-1,b35);
array_bool_or([b32,b35],true);
int_lin_le(i10,[i3,i6],0);
int_lin_le_reif(i10,[i4,i7],-1,b38);
array_bool_or([b33,b38],b39);
int_lin_le_reif(i10,[i4,i7],0,b40);
int_lin_le_reif(i10,[i5,i8],-1,b41);
int_lin_le_reif(i10,[i5,i8],0,b43);
array_bool_and([b53,b54],b46);
array_bool_and([b55,b57],b47);
int_lin_le_reif(i10,[i1,i0],-1,b49);
array_bool_or([b46,b49],true);
int_lin_le(i10,[i1,i0],0);
int_lin_le_reif(i10,[i4,i3],-1,b52);
array_bool_or([b47,b52],b53);
int_lin_le_reif(i10,[i4,i3],0,b54);
int_lin_le_reif(i10,[i7,i6],-1,b55);
int_lin_le_reif(i10,[i7,i6],0,b57);
array_bool_and([b67,b68],b60);
array_bool_and([b69,b71],b61);
int_lin_le_reif(i10,[i2,i1],-1,b63);
array_bool_or([b60,b63],true);
int_lin_le(i10,[i2,i1],0);
int_lin_le_reif(i10,[i5,i4],-1,b66);
array_bool_or([b61,b66],b67);
int_lin_le_reif(i10,[i5,i4],0,b68);
int_lin_le_reif(i10,[i8,i7],-1,b69);
int_lin_le_reif(i10,[i8,i7],0,b71);
solve satisfy;

```

Listing 6.4: FlatZinc for Latin Squares with $n=3$ with 42 variables and 58 constraints.

swapped. In the latter, the `lex_less` constraint is decomposed introducing several temporary constants (`lx`, `ux`, `ly`, `uy`, and `size`), which make the body of the constraint more succinct. In addition, it introduces an array of Boolean variables `b`, representing whether or not each pair of corresponding elements in the arrays are ordered, with one extra element (`b[size + 1]`) to represent whether the first array is shorter than the second. The body defines a set of relational constraints which define these Booleans, the first of which (`b[0]`) must be true. Listing 6.4 shows the output from the MiniZinc compiler when compiling this model with $n = 3$. For brevity, the `constraint` keyword has been omitted. We can see from this FlatZinc program that, once a model has been compiled, it can be difficult for a user, especially a novice one, to interpret and map these constraints back to their source in the original model.

6.3 Related Work

This section discusses existing MUS-based approaches for debugging constraint problem specifications. A Minimal Unsatisfiable Subset (MUS) is an unsatisfiable set of constraints where the removal of any one constraint makes the set satisfiable. A MUS, being minimal, attempts to succinctly describe a source of unsatisfiability in a constraint program. A related concept is that of a Maximal Satisfiable Subset (MSS), that is, a satisfiable subset of constraints, where the addition of any extra constraint from the full program makes the set unsatisfiable.

Typical approaches for fault diagnosis of constraint problem specifications work at the constraint program level. These approaches aim at enumerating all (or some subset of) the MUSes that can point to the sources of unsatisfiability. Having a selection of MUSes gives the user a better chance to discover the root cause of unsatisfiability, although in some cases a single MUS may be sufficient.

Enumerating all of the MUSes of a program can be achieved by exploring the power-set, that is, all possible subsets of constraints, to perform a satisfiability check on each, and collect the unsatisfiable ones, discarding all strict supersets. The satisfiability check is typically delegated to an external solver, with the algorithm just operating on a set of labels (typically integers) representing the program level constraints. This allows a MUS detection algorithm to be agnostic in the concrete type of problem that is being diagnosed.

Most MUS algorithms avoid enumerating the entire power-set, attempting instead to minimise the number of satisfiability checks required by pruning the set intelligently. For example, by not exploring any superset of an already discovered MUS. A detailed survey of MUS enumeration approaches can be found in Liffiton et al. (2015). These approaches work with the full set of program level constraints (such as all 58 constraints in Listing 6.4). Good techniques for pruning the search space can reduce the number of satisfiability checks considerably. However, depending on the time taken by each satisfiability check, the pruning may not be enough to make these approaches practical for large programs.

As mentioned above, there has been much work in the area of program level MUS enumeration. Some approaches focus on the characteristics of specific constraint systems in order to be able to further prune the search space. For example, for mixed integer linear programming, properties of the linear system can be taken into account (Gleeson et al., 1990; Loon, 1981). For unsatisfiable numerical CSPs (NCSPs), algorithms exploiting structure inherent to NCSPs to prune the power-set of program constraints have also been proposed (Gasca et al., 2007). While powerful, these methods restrict the applicability of the debugging methods and are, thus, not the focus of this thesis.

In addition, there are several algorithms available for constraint-system agnostic MUS enumeration (Bacchus et al., 2016; Bailey et al., 2005; Liffiton et al., 2013, 2015). Many of these algorithms focus on finding explanations (in the form of MUSes) during search, as the number of applicable MUSes is then much smaller. However, these methods can also be used for diagnosing unsatisfiable constraint programs, if scalability can be achieved. One approach, known as QuickXplain (Junker, 2001) attempts to discover MUSes using a divide and conquer approach. It works by removing a constraint from the full set, and

recursively enumerating MUSes. Once enumerated, it recursively explores the set of constraints that include this constraint. Later approaches such as Dualize and Advance (DAA) from Bailey et al. (2005) use more powerful techniques for pruning the search space. DAA exploits the duality between Minimal Correction Sets (sets that, when removed completely from unsatisfiable sets makes them satisfiable, and removing only a subset of them leaves them unsatisfiable) and MUSes. Hitting sets are computed for each MCS to discover MUSes. The main drawback of this approach was that the enumeration of the hitting sets incurs a prohibitive memory and time cost. The MARCO algorithm (Liffiton et al., 2013) and the more recent MCS-MUS-BT (Bacchus et al., 2016), provide much more efficient approaches for finding MUSes. Details of the MARCO algorithm, which is the one used in our implementation, are presented in Section 6.3.

Existing MUS enumeration approaches are likely to produce MUSes that include constraints and variables introduced during compilation. It can be difficult for a user to map these back to the source in the model. Thus, presenting these MUSes to a user who is unfamiliar with the workings of the compiler may not actually be conducive to fixing modelling mistakes. Consider the final program in Listing 6.4, which has a large set of `int_line` constraints. Deciding which call to `alldifferent` these constraints come from in the original model is possible but can be difficult. Recall that MiniZinc does not guarantee the constraints are inserted into the FlatZinc program in the order they are encountered. One example of this out-of-order compilation is when the compiler encounters a constraint in a reified context. The compiler will introduce a Boolean control variable for the constraint and then defer the compilation of the constraint until a later point. If the control variable is fixed to true later during compilation a non-reified version of the constraint can be used, leading to a more efficient constraint program. This becomes more important when a target solver does not support a reified version of a global constraint, where, without the out of order compilation, the constraint would have to be decomposed.

At the modelling system level there has been some effort to provide more meaningful explanations of unsatisfiability. In Jussien et al. (2001) users can explicitly group their constraints, giving a user friendly name to each sub-group. When a constraint system is found to be unsatisfiable, explanations are found at the system level (program level). These are then mapped to the names provided by the user to provide better feedback. Our approach builds on this idea, but shifts the burden from the user to the debugging system. CPTEST (Lazaar et al., 2012) is a modelling system level framework that can aide a user in correcting several types of faults in a model given a correct (though possibly naive) model as an “oracle”. It focusses on finding faults involving individual model constraints. It finds these by negating each one, and compares the resulting search space of the model with that of the oracle. Unfortunately, some problems are difficult to model even naively. Thus, non-oracle based approaches are indeed needed.

The MARCO Algorithm

Our approach to structure guided fault diagnosis is based on the use of a constraint-system agnostic MUS algorithm. In particular, the proof-of-concept implementation evaluated in

Algorithm 6.1 An implementation of `EnumerateMUSes` adapted from the most basic form of the MARCO algorithm from Liffiton et al. (2015).

```

function MARCO(labels, maxMUSes)
  MUSes  $\leftarrow \emptyset$  ▷ Set of sets of labels
  map  $\leftarrow$  BooleanFormula(vars = labels, clauses =  $\emptyset$ )
  while map is satisfiable and  $|MUSes| \leq maxMUSes$  do
    activeLabels  $\leftarrow$  GetUnexplored(map)
    if check(activeLabels) then
      MSS  $\leftarrow$  Grow(activeLabels)
      map  $\leftarrow$  map  $\wedge$  BlockDown(MSS) ▷ Block all subsets of MSS
    else
      MUS  $\leftarrow$  Shrink(activeLabels)
      map  $\leftarrow$  map  $\wedge$  BlockUp(MUS) ▷ Block all supersets of MUS
      MUSes  $\leftarrow$  MUSes  $\cup \{MUS\}$ 
  return MUSes

```

Algorithm 6.2 Satisfiability check.

```

function check(activeLabels)
  M  $\leftarrow \emptyset$  ▷ Start with empty constraint program
  for each (p  $\in$  activeLabels) do
     $\langle d, C \rangle \leftarrow T[p]$ 
    M  $\leftarrow M \cup \bigwedge_{c \in C} c$  ▷ Assume variables are added automatically
  return solve(M)

```

this Chapter uses the MARCO algorithm. MARCO was chosen due to its simplicity and the availability of an existing implementation that could be extended. An adapted version of the basic MARCO algorithm is reproduced from Liffiton et al. (2015), in Algorithm 6.1. Here, the algorithm takes the form of a function `MARCO(labels, maxMUSes)` which implements an interface, `EnumerateMUSes`, which the new approach requires. The interface must take a set of constraint labels, *labels*, and a maximum number of MUSes, *maxMUSes*, to enumerate. It returns a set of MUSes defined in terms of the labels. The MARCO algorithm can be used to both find minimal unsatisfiable sets (MUSes) and Maximal Satisfying Sets (MSSes). The algorithm selects a subset of constraints, and checks if it is satisfiable. If it is, it grows the set by adding constraints that do not cause unsatisfiability until it is an MSS. Alternatively, if the selected subset is unsatisfiable, it removes constraints that do not contribute to the unsatisfiability. Once an MSS or MUS has been found, it then selects a new subset that is not a subset of an MSS or a superset of a MUS.

To achieve this the algorithm maintains a CNF formula called *map*. The *map* initially comprises a `BooleanFormula`, initialised with variables representing the selection of certain constraints (by label) in a program, and an empty set of clauses. That is, any subset can be selected. The *map* is used as a generator of subsets that have not yet been checked.

The main loop of the algorithm executes until the *map* is no longer satisfiable, signifying that the entire power-set of constraints has been explored, or until the number of enumerated MUSes reaches *maxMUSes*. The algorithm calls `GetUnexplored`, which requests a satisfying solution to the *map* and stores it as *activeLabels*. The *activeLabels* are

then passed to `check` (Algorithm 6.2), which returns whether or not the subset is satisfiable. If the set of constraints is satisfiable, it is *grown* by calling `Grow`. This procedure incrementally grows the *activeLabels* set into an MSS from the set *labels*, checking for satisfiability as it goes. In particular, if adding a label to *activeLabels* causes unsatisfiability, it is removed and the procedure continues with the remaining constraints. This set is then returned and stored in *MSS*. A clause is then added to the *map* to reflect that all strict subsets of *MSS* are satisfiable and do not need to be explored. This is accomplished by using the procedure `BlockDown`, which returns the clause $\bigvee_{i \notin MSS} x_i$, where x_i is a label from the full set of *labels*. Alternatively, if the subset was unsatisfiable, the `Shrink` procedure is called to remove constraints from the *activeLabels* set that do not appear to cause unsatisfiability. Thus, `Shrink` returns a MUS. In this case a clause is added to the *map* that marks all supersets as explored. This is achieved using the procedure `BlockUp`, which returns the clause $\bigvee_{i \in MUS} \neg x_i$. The algorithm continues as long as *map* is satisfiable, indicating that unexplored subsets remain.

6.4 Exploiting Model Structure for MUS Detection

This section introduces the new approach for guiding an existing MUS detection algorithm to find MUSes using the explicit high-level structure present in a model. The approach builds upon two existing techniques and extends the MiniZinc variable path concept, in order to speed up MUS detection and provide more meaningful diagnoses to the user. The first existing technique is that of Jussien et al. (2001), where explanations, found at the program level, are described with user friendly names for grouped constraints. These names, which are manually obtained from the user encode the constraint hierarchy. This is used to provide more readable, useful feedback. Our method extends this concept to show how existing model-level structure can be used to group constraints *automatically*, by using the concept of *constraint paths*, which are an extension of the variable paths introduced in Chapter 4. Further, we show how constraint paths allow for an iterative refinement of MUSes that avoids the complete exploration of the space. The second existing technique is that of constraint agnostic MUS enumeration. As indicated before, the prototype implementation uses MARCO. However, any MUS enumeration algorithm that implements the `EnumerateMUSes` interface could be used.

6.4.1 Constraint Paths

In Jussien et al. (2001) the modeller is tasked with manually grouping related constraints together. One of the key contributions of this chapter is the insight that a high-level language, like MiniZinc, has by default an explicit hierarchy of constraints and, therefore, does not need to be annotated by a user. To access this hierarchy at the program level, we need a mechanism that preserves this hierarchical structure through compilation.

The concept of *variable paths* was presented in Chapter 4. Recall that variable paths describe the path the compiler took through the model to the point where a concrete variable is introduced to the program. Paths contain information on the location of each

```

30  int_lin_ne(i10, [i3,i4],0);      5:12:forall:i=2 all_different_int forall:i=1 j=2 !=
31  int_lin_ne(i10, [i3,i5],0);      5:12:forall:i=2 all_different_int forall:i=1 j=3 !=
32  int_lin_le(i10, [i3,i6],0);
    11:12:forall:r=2 lex_less lex_less_int let /\ forall:i=0 = /\ forall <=
33  int_lin_le_reif(i10, [i4,i7], -1,b38);
    11:12:forall:r=2 lex_less lex_less_int let /\ forall:i=1 = /\ forall \/\ exists <

```

Listing 6.5: Simplified MiniZinc paths with depth 1 prefix marked in bold.

syntactic construct, as well as the bindings of all loop variables involved in producing a variable. In Chapter 4 we used paths to match variables across multiple compilations of an instance. Our fault diagnosis approach requires the development of a similar path concept for constraints (in addition to that of variables) to access the extra structure available in the form of the hierarchy of constraints, and also to be able to tie low-level constraints in a MUS back to the user’s model. The construction of constraint paths can be added to the reduced MiniZinc compiler presented in Chapter 2, by simply modifying the `postConstraints` procedure to call `getCurrentPath` when constraints are being added to the FlatZinc, and by adding the path as an annotation to the constraint. Pseudocode for this can be seen in Algorithm 6.3.

Algorithm 6.3 Annotating program constraints with paths.

```

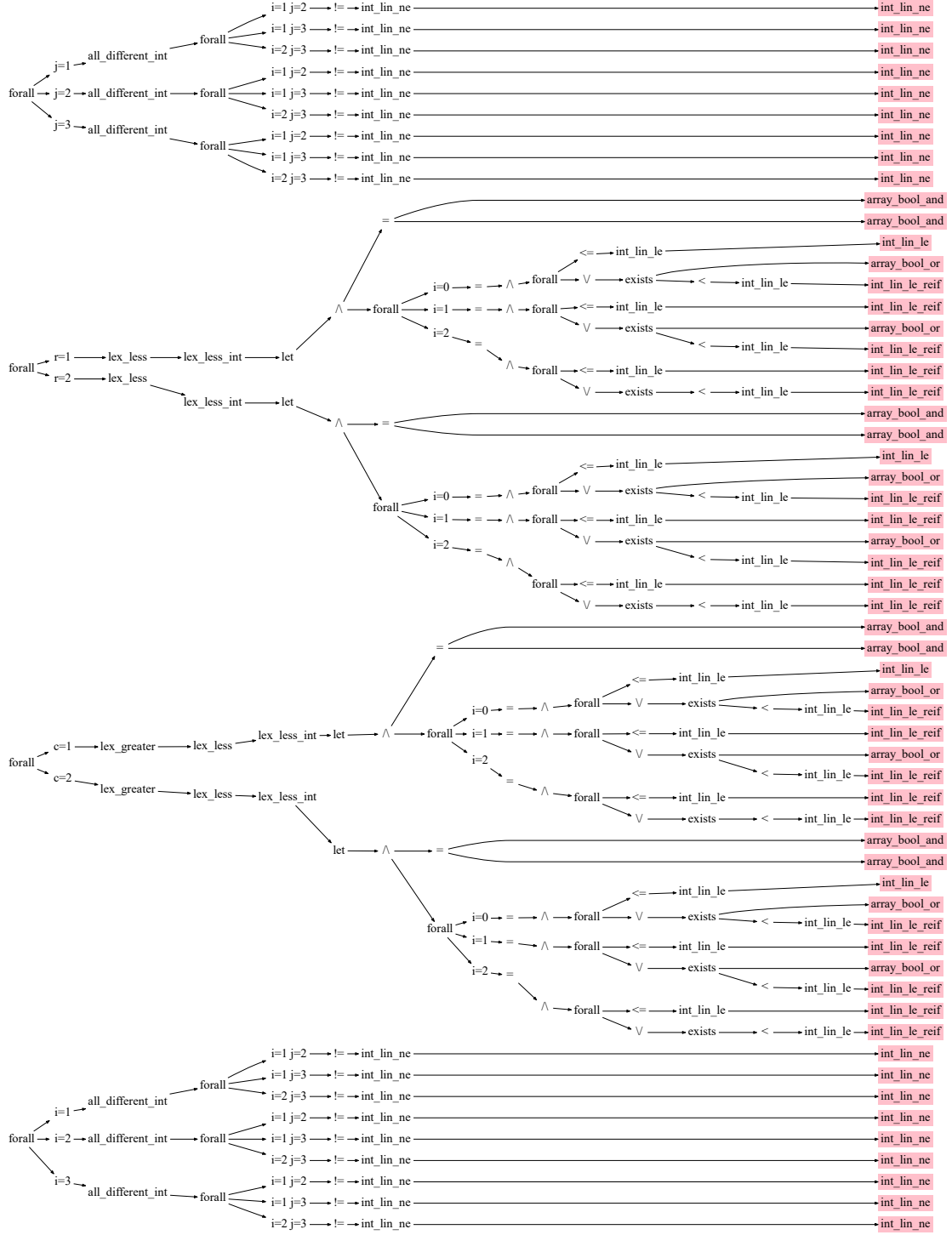
procedure postConstraints(C)
  for each (c ∈ C) do
    path ← getCurrentPath()
    annotations(c) ← annotations(c) ∪ {path}
    S ← S ∪ {c}

```

Figure 6.1 and Listing 6.5 illustrate how paths for constraints can expose the hierarchy of constraints at the program level. This section will show how the variable paths approach from Chapter 4 can be extended to make this possible. Listing 6.5 shows simplified paths for four constraints from the compiled Latin Squares instance where $n=3$, which was already shown in Listing 6.4. The path for the third constraint (`int_lin_le(i10, [i3,i6],0)`) shows that it was introduced by the `lex_less` constraint for row $r=2$. This means it was introduced by call `lex_less(row(X, 2), row(X, 2+1))`. Following the rest of the path, we can see that this specific `int_lin_le` constraint can be traced to the less-than (`<`) on line 14 of the decomposition presented in Listing 6.3. As in Chapter 4, the amount of detail in the paths presented here has been reduced for illustrative purposes. Actual MiniZinc paths retain information about what file a call is in, and the span of text the call covers (start and end rows and columns).

Figure 6.1 presents the trace of compilation for all constraints in the flattened program of Listing 6.4. The figure clearly shows that there is a hierarchy of constraints, where a `forall` introduced a set of `all_different_int` constraints, and the decomposition of these introduced sets of individual `int_lin_ne` constraints. Note also how the `lex_greater` constraints are clearly seen as being replaced with `lex_less` constraints.

The figure also shows how paths make both the model-level (what the user wrote) hierarchy and the decomposition (or compiler-level) hierarchy, available at the program

Figure 6.1: A reduced trace tree for the Latin Squares model with $n=3$.

level. At the root of the tree there are four different high-level constraints (the four `forall` loops). Descending to a depth of 1 exposes the two sets of three `all_different` constraints on the rows and columns, and the two pairs of `lex` constraints also on the rows and columns. Descending deeper exposes the decomposition of these constraints into the individual program level constraints (the `int_lin_ne` constraints coming from the decomposition of `all_different`, and the `int_lin_le_reif` constraints from `lex_less`).

6.4.2 Grouping Constraints by Paths

Having access to the model-level structure at the program-level allows constraints to be grouped together automatically when searching for MUSes. Our method combines the advantages of not having to rely on the user to define the constraint groups, with those of not having to explore the full set of program level constraints. It achieves this by generating automatically the constraint groups based on the model structure, starting from the top-level constraint items, and iteratively refining them down to individual program-level constraints only if needed. This allows for a novel approach that is scalable, automatic, and can produce more accessible diagnoses.

The approach operates on a reduced form of the hierarchy where only nodes that have multiple child nodes are included, that is, it only contains branching nodes. The root node of the hierarchy presented in Figure 6.1 (or depth 0, which is omitted from the diagram) for the Latin Squares model, would have 4 constraint groups corresponding to the top-level constraints from lines 5, 7, 11, and 13 in the original model. Grouping constraints under branching nodes in this hierarchy, and treating them as a single unit for the purposes of MUS enumeration, allows us to find “high-level” MUSes.

Take for example Figure 6.1. Grouping constraints by their path prefixes to a depth of 0, a MUS enumeration algorithm will have to explore the power-set of only four constraint groups, which is a much smaller search space than that of the full 58 program level constraints. Of course, MUSes found at the root can only hint at what might be wrong, by drawing a user’s attention to specific lines in the model. More fine-grained and, thus, more useful MUSes, will have to be found deeper in the hierarchy by using longer prefixes.

With this approach, the mapping from labels representing constraint groups to the individual program-level constraints that they represent, must be recorded. Algorithm 6.4 presents the function `split`, which takes a set of constraints C , a depth $depth$, and a map T' . This map maintains a mapping from a label (a path prefix of the first d nodes in a path) to a 2-tuple $\langle depth, cons \rangle$, which records the depth of a prefix and the set of program-level constraints that share this prefix, grouping them as a single constraint group. Note that the algorithms here make use of *ReversePathMap*. This map was introduced in Chapter 4 for managing the unification of variables, but can also be used to provide a mapping from program-level constraints back to their respective paths. The `split` function iterates through each constraint c in the set of constraints C . For each such c , it gets a path from *ReversePathMap* and uses the `prefix` function to select the first $depth$ elements of the path to be used as the group label. If T' has an entry corresponding to this label, the constraints of that entry are extracted as $cons$. Finally, the entry in T' for the label p is set (or updated) with the new constraint added to $cons$ ($cons \cup \{c\}$).

Selective deepening. Algorithm 6.5 presents the an algorithm for finding MUSes. The `diagnose` procedure takes as arguments a set of constraints C , a minimum depth to begin searching for MUSes, $minDepth$, a depth at which to stop searching, $maxDepth$, a maximum number of MUSes to discover at each depth, $maxMUSes$, and a flag, *complete* stating whether constraint groups can be discarded during the search. The process starts

Algorithm 6.4 Function that groups constraints that share a prefix.

Global State:*ReversePathMap*: map: *expression* \rightarrow *path***Parameters:***C*: set of constraints ▷ Program level constraints*depth*: integer ▷ Depth of prefixes*T'*: map: *label* \rightarrow $\langle \text{depth}, \text{cons} \rangle$ ▷ Maps labels to associated constraints**function** *split*(*C*, *depth*, *T'*) **for each** (*c* \in *C*) **do** $\text{cons} \leftarrow \emptyset$ ▷ Get first *depth* elements from path of *c* $p \leftarrow \text{prefix}(\text{ReversePathMap}[c], \text{depth})$ **if** $p \in T'$ **then** ▷ If prefix is already in *T* use existing *cons* $_, \text{cons} \leftarrow T'[p]$ $T'[p] \leftarrow \langle \text{minDepth}, \text{cons} \cup \{c\} \rangle$ **return** *T'*

Algorithm 6.5 Procedure for reporting MUSes found at different depths.

Global State:*T*: map: *label* \rightarrow $\langle \text{depth}, \text{cons} \rangle$ ▷ Maps labels to associated constraints**Parameters:***C*: set of constraints ▷ Program level constraints*minDepth*: integer ▷ Starting depth*maxDepth*: integer ▷ Depth to stop at*maxMUSes*: integer ▷ Maximum number of MUSes to find at each depth*complete*: Boolean ▷ Should constraints be discarded?**procedure** *diagnose*(*C*, *minDepth*, *maxDepth*, *maxMUSes*, *complete*) $T \leftarrow \text{split}(C, \text{minDepth}, \emptyset)$ **do** $\text{MUSes} \leftarrow \text{EnumerateMUSes}(\text{labels of } T, \text{maxMUSes})$ $\text{report}(T, \text{MUSes})$ **while** $\neg \text{increase_depth}(\bigcup \text{MUSes}, \text{maxDepth}, \text{complete})$

by initialising the global map *T* by calling the *split* function with the full set of constraints *C*, the starting depth, *minDepth*, and an empty map. *T* essentially represents the frontier of the diagnosis algorithm. A do-while loop calls a MUS enumeration algorithm that implements the *EnumerateMUSes* interface with the labels of *T*, that is, only the labels of groups of constraints, not the constraints themselves, and *maxMUSes* which states the maximum number of MUSes we wish to retrieve at this depth. The returned MUSes are then presented to a user by the call to *report*.

It is important to note that a “MUS” that includes constraint groups, rather than individual program level constraints, does not correspond directly to a single program level MUS. After a MUS at a certain depth is found, there is guaranteed to be at least one MUS at the program level involving a subset of the program constraints represented by the constraint groups in the MUS, but possibly even more.

To find these (giving more fine-grained information about the MUS) we must first split the constraint groups by increasing the depth as described in Algorithm 6.6. This could be

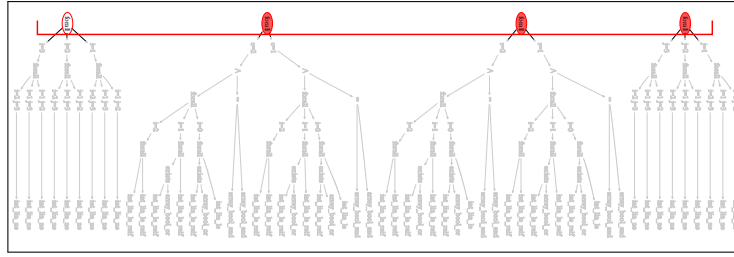
Algorithm 6.6 Mapping *labels* to deeper constraint groups.**Global State:** T : map: $label \rightarrow \langle depth, cons \rangle$ ▷ Maps labels to associated constraints**function** `increase_depth(labels, maxDepth, complete)` $T' \leftarrow \emptyset$ **for each** $(p \in labels)$ **do** $\langle d, C \rangle \leftarrow T[p]$ $d' \leftarrow \text{Min}(d+1, maxDepth)$ $T' \leftarrow \text{split}(C, d', T')$ **if** *Complete* **then**▷ Retain constraint groups not in *labels***for each** $(p \notin labels)$ **do** $T'[p] \leftarrow T[p]$ $T \leftarrow T'$ **return** can the prefixes be extended? ▷ Are the prefixes all at their *maxDepth*

done by simply increasing the depth for all constraint groups, splitting each of them again according to the next part of their respective paths. However, this would be inefficient, as the depth does not need to be increased for constraint groups that did not take part in any MUS. Only expanding the groups that are involved in a MUS allows the algorithm to avoid a detailed search through constraints that are likely irrelevant and to spend more time focussing on constraints that are likely to be involved in a MUS at this increased depth. This can speed up enumeration considerably, since a constraint group that is not involved in any MUS can potentially contain a large set of constraints. In Algorithm 6.5 this is represented by the call to the `increase_depth` procedure with the union of constraint groups that occur in the discovered MUSes.

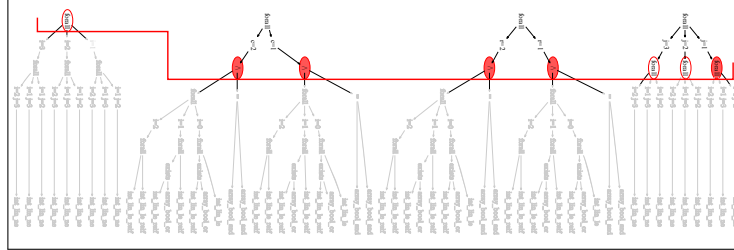
The `increase_depth` procedure, defined in Algorithm 6.6, takes a set called *labels* representing the set of constraints to be split, a limit *maxDepth* to increase the depth to, and a flag, *complete* which states whether the algorithm can discard constraint groups. The purpose of this procedure is to construct a new T' that results from unpacking the constraint groups from T that correspond to the labels in *labels*. It calculates new labels based on longer prefixes of their constituent constraint paths. It then builds a new T' with these labels. This splits the constraint groups into a set of smaller constraint groups that are from deeper in the hierarchy. If the *complete* argument is set to true, the algorithm adds to T' the set of labels and associated constraint groups that were not in *labels*. Finally, it replaces the original T with T' . This makes sure that all program constraints remain reachable. It then returns a Boolean stating whether the *maxDepth* has been reached.

If `increase_depth` returns *true*, the `diagnose` procedure is finished. Calling `increase_depth` on the union of found MUSes, essentially implements a kind of breadth-first search for MUSes. To allow the search for a MUS to be more focussed, *maxMUSes* can be set to 1, forcing the MUS enumeration to only find a single MUS at each depth before going deeper. An evaluation of this approach is presented in Section 6.4.3.

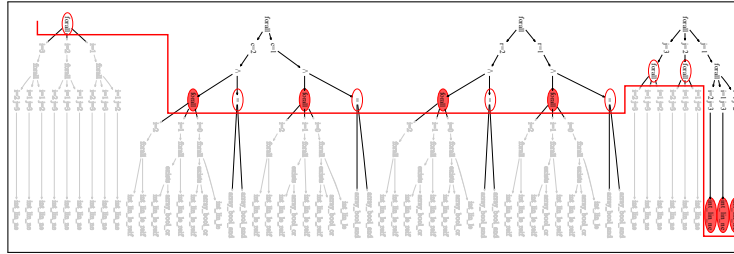
To illustrate how the selective deepening approach works, consider a simple example where the approach is applied to a set of constraint groups labelled $\{1, 2\}$. Assume we find the MUS $\{1\}$. This means the depth only needs to be increased for constraint 1,



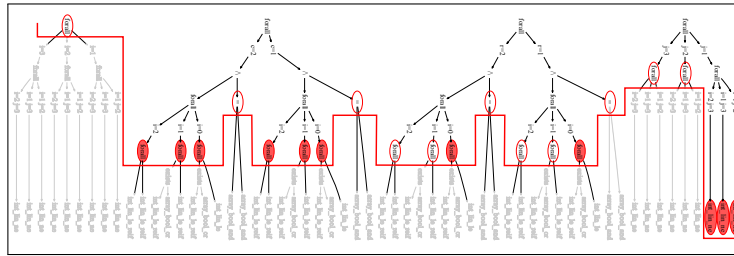
(a) Frontier: 4 constraint groups.



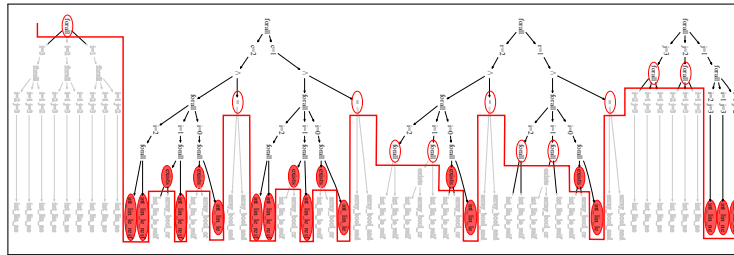
(b) Frontier: 8 constraint groups.



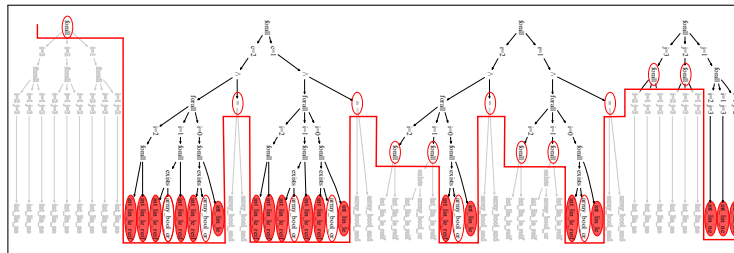
(c) Frontier: 14 constraint groups.



(d) Frontier: 22 constraint groups.



(e) Frontier: 30 constraint groups.



(f) Frontier: 36 constraint groups.

Figure 6.2: Selective Deepening.

which may result, for example, in the new set $\{1.1, 1.2, 1.3, 2\}$. If the second constraint group contains five lower level constraints, taking this approach leads to a significant reduction in the number of subsets to be explored over the number of possible subsets in: $\{1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 2.4, 2.5\}$.

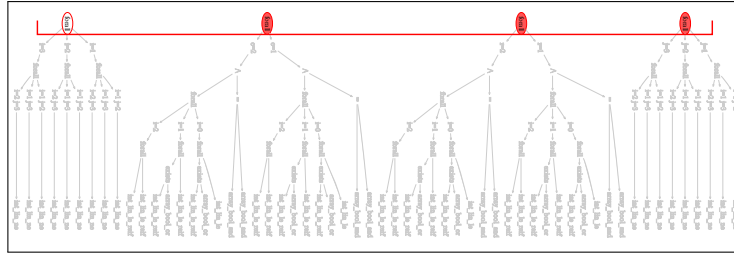
A more detailed demonstration of the workings of the algorithm is presented in Figure 6.2. The different panels of this figure show the progression of the frontier of the algorithm as it is applied. The red ellipses (filled in or not) mark the branching points that are being explored by the algorithm. The filled in, red ellipses mark MUSes that have been found. The red line highlights the frontier. The trace of paths presented here is a reduced version of the compiled Latin Squares constraint program presented in Figure 6.1. The first iteration of the algorithm calls the MUS enumeration algorithm on 4 constraint groups, finding a MUS involving 3 of them. Increasing the depth for these 3 constraints results in the frontier growing to 8 constraints. Once program level constraints are reached at depth 6, a set of only 36 constraints needs to be explored instead of the full set of 58.

Incomplete enumeration. A further optimisation that can improve scalability is to omit entirely the constraint groups that did not occur in any MUS at the current depth. Figure 6.3 shows how the frontier is focussed while searching for a MUS in the Latin Squares example. The difference is evident after only the first call to `increase_depth`, which leaves the frontier with only 7 constraint groups on the frontier, improving slightly on the 8 constraint groups that the complete approach finds at this stage. As search continues we see that the difference becomes even more apparent, with the incomplete approach only having to look at 25 constraints at the program-level, compared with the 36 of the complete approach. This incomplete deepening approach can allow us to quickly discover MUSes. However, the removal may occasionally cut off MUSes that can only be found at lower depths.

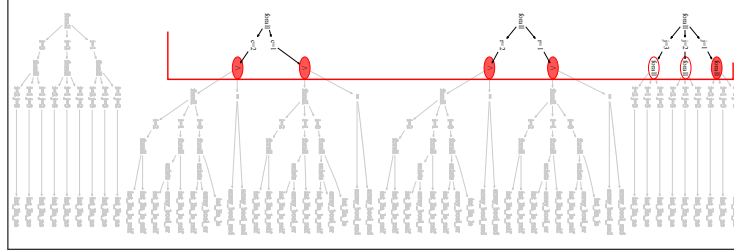
To demonstrate how incomplete deepening can omit valid MUSes, Figure 6.4 shows a simple faulty model where incomplete deepening will miss a valid program level MUS. This example has two MUSes, shown in Figure 6.4b and Figure 6.4c: $\{x < 4, x > 5\}$ and $\{x > 6, x < 4\}$, respectively. A MUS detection algorithm searching at a depth of 0, i.e., only looking at each constraint item as a whole, will find the conflict $\{x > 6, x < 4 \wedge x > 5\}$. This set is not minimal, as the set is still unsatisfiable when constraint $x > 6$ is removed. Incomplete enumeration will then seek to find MUSes in the set $\{x < 4, x > 5\}$ discarding constraint $x > 6$ and, thus, cutting off the MUS $\{x > 6, x < 4\}$. This approach sacrifices completeness for scalability, allowing users to quickly discover some subset of the MUSes in their model. In Algorithm 6.5 this is implemented by passing *false* as the *complete* argument to `increase_depth`, which implements incomplete enumeration by including or omitting labels that are not included in a MUS from the new T .

6.4.3 Implementation Details

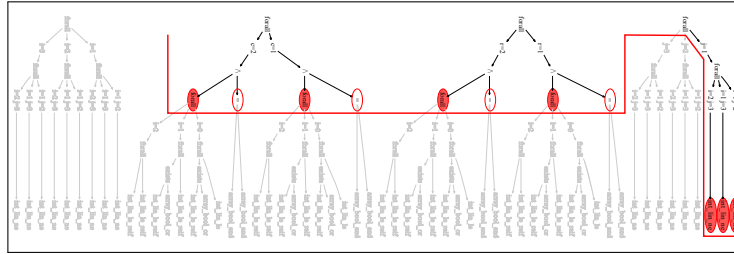
The proof of concept implementation evaluated in the next section is based on the released code for the MARCO approach. The approach was implemented as a Python script with



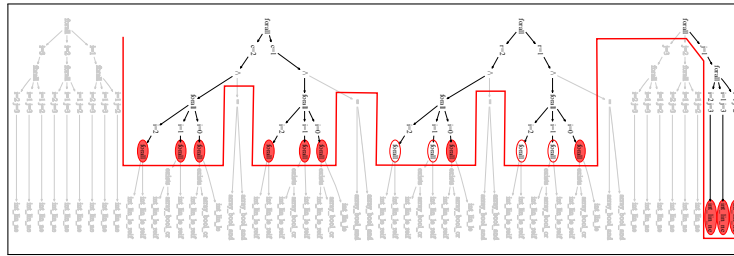
(a) Frontier: 4 constraint groups.



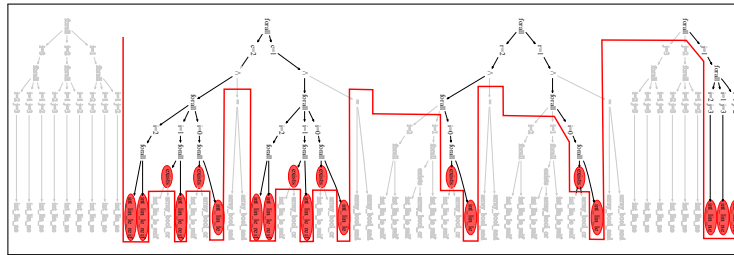
(b) Frontier: 7 constraint groups.



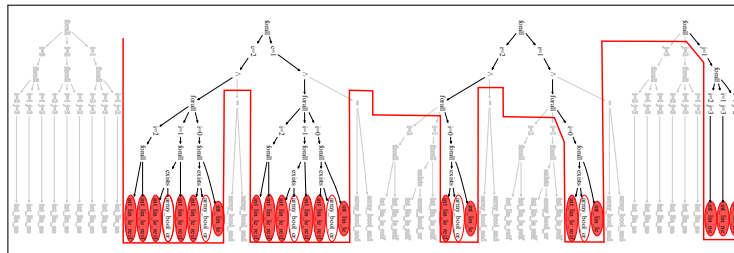
(c) Frontier: 11 constraint groups.



(d) Frontier: 15 constraint groups.



(e) Frontier: 19 constraint groups.



(f) Frontier: 25 constraint groups.

Figure 6.3: Incomplete Deepening.

```

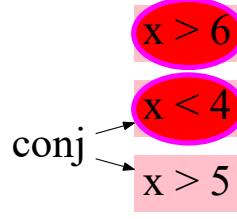
% simple.mzn
var int: x;

x > 6;
x < 4 /\ x > 5;

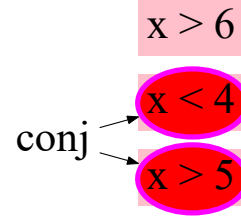
solve satisfy;

```

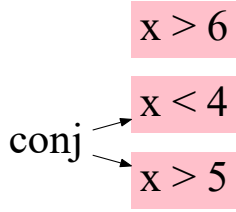
(a) Example Model.



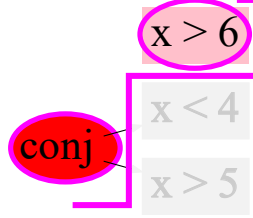
(b) MUS 1.



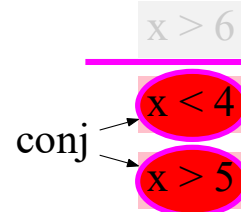
(c) MUS 2.



(d) Compilation Trace.



(e) Conjunction is minimal.



(f) MUS 1 discarded.

Figure 6.4: Incomplete enumeration can miss some MUSes.

interfaces to some SAT solvers, including MiniSAT and MiniCard. The script allows for new satisfaction checkers to be integrated relatively easily. The core MARCO algorithm represents the selection of different constraints as simple integers. Our implementation added a satisfaction checker for FlatZinc that is aware of MiniZinc paths and of the grouping of constraints. In addition, the implementation included a modification to the MARCO frontend that implements the algorithm presented in Algorithm 6.5.

The frontend supports running MARCO on a set of constraint groups at a target depth, by setting *minDepth* and *maxDepth* to the same value. To perform selective deepening instead, *minDepth* must be less than *maxDepth*. In this case MUSes will be enumerated at *minDepth*, with *increase_depth* being called until the frontier cannot be expanded any further, or until *maxDepth* is reached. Incomplete enumeration occurs when *complete* is set to *false*. The number of MUSes discovered by each call to *EnumerateMUSes* can be configured by setting *maxMUSes*. This setting, combined with *complete* being set to *false*, allows the tool to focus on finding the *maxDepth* constraint groups related to a specific fault as quickly as possible.

6.5 Experimental Evaluation

To evaluate the efficiency and usefulness of the new approach, two experiments were performed. The first explores the effectiveness of the approach for finding multiple MUSes at different depths. The second experiment explores how it behaves when only searching for a single MUS. Six different configurations of the system were evaluated, paired by their target *maxDepth*. Each execution was given a timeout of 300 seconds (5 minutes). The configurations are presented in Figure 6.5. Configurations C_2 , C_3 , and $C_{Program}$ correspond to a fixed depth MUS enumeration at depths 2, 3, and the depth of the individual program-level constraints. This program-level enumeration corresponds to the

Config	<i>minDepth</i>	<i>maxDepth</i>	<i>complete</i>
C_2	2	2	<i>true</i>
$C_{1 \rightarrow 2}$	1	2	<i>false</i>
C_3	3	3	<i>true</i>
$C_{1 \rightarrow 3}$	1	3	<i>false</i>
$C_{Program}$	Program	Program	<i>true</i>
$C_{1 \rightarrow Program}$	1	Program	<i>false</i>

Figure 6.5: Experiment configurations.

traditional application of MARCO. Configurations $C_{1 \rightarrow 2}$, $C_{1 \rightarrow 3}$ and $C_{1 \rightarrow Program}$ perform incomplete enumeration (setting *complete* to *false*), enumerating MUSes first at a depth of 1 and then increasing the depth to 2, 3, and the program depth, respectively, discarding constraint groups that are not involved in any MUS as they proceed.

Dataset

Currently, there are no collections of constraint models that contain bugs that make them unsatisfiable. Collections such as CSPLib (Gent et al., 1999) or the MiniZinc benchmarks¹ contain only finished, correct models for problems. We therefore introduced artificial faults into some of the models from the MiniZinc Challenge 2015 (Stuckey et al., 2010), similarly to how fault injection was applied in (Lazaar et al., 2012). The faults were introduced so that each model became unsatisfiable in a non-trivial way, i.e., so that the compiler does not detect it already during compilation and the solver must perform some search to prove unsatisfiability. The instances for each model were selected at random. The following lists the models used, along with the faults that were injected. The source for these faulty models can be found in Appendix C.

Costas-Array: This problem can be seen as a two dimensional version of the Golomb ruler problem. Points must be placed in an $n \times n$ matrix, such that the displacement vectors between each pair of points are distinct. The fault added is a common off-by-one mistake in the indexing of an array: `array access differences[k-2, l-1]` became `differences[k-2, l-2]`.

CVRP : A capacitated vehicle routing problem. This is the classic VRP with maximum capacities for the vehicles. The fault injected replaces the use of a successor variable array in one of the constraints with the predecessor variable array: expression `successor[n]` became `predecessor[n]`.

Free-Pizza: The goal of this problem is to find the best way to use a given set of vouchers for acquiring pizzas. The fault added replaces a *less-equal* with a strict *less-than*: constraint `<=` became `<`.

Mapping : This problem is about mapping streaming applications on multiprocessors with time-division-multiplexed network-on-chip. The added fault changes a constant in an expression from a zero to a one.

¹<https://github.com/MiniZinc/minizinc-benchmarks>

Configuration	C_2			$C_{1 \rightarrow 2}$			C_3			$C_{1 \rightarrow 3}$			$C_{Program}$			$C_{1 \rightarrow Program}$		
Model	T	G	M	T	G	M	T	G	M	T	G	M	T	G	M	T	G	M
Costas-Array	300.0	56	88	300.0	54	11	300.0	83	0	300.0	0	0	300.0	83	0	300.0	0	0
CVRP	300.0	34	1	0.5	7	1	300.0	66	2	0.7	7	1	300.0	101	3	1.4	14	4
Free-Pizza	300.0	81	13	0.7	9	1	300.0	82	12	0.7	1	1	300.0	553	8	1.2	1	1
Mapping	0.4	24	1	0.3	2	1	28.9	69	70	10.9	28	66	300.0	254	70	300.0	0	0
MKnapsack	7.6	31	49	7.2	30	49	7.8	31	49	14.3	30	49	9.5	32	65	58.2	31	65
NMSeq	300.0	40	0	300.0	40	0	300.0	40	0	300.0	0	0	300.0	3240	0	300.0	0	0
Open-Stacks	300.0	802	5	300.0	800	5	300.0	841	4	300.0	0	0	300.0	4421	0	300.0	0	0
P1F	113.6	89	77	89.3	77	77	113.9	89	77	170.6	77	77	300.0	947	10	300.0	0	0
Radiation	0.5	5	1	0.5	4	1	12.0	68	12	4.1	25	12	73.1	388	12	20.4	12	12
Spot5	300.0	4998	0	300.0	4406	1	300.0	5227	0	300.0	0	0	300.0	5457	0	300.0	0	0
TDTSF	300.0	46	1	300.0	0	0	300.0	62	1	300.0	0	0	300.0	170	1	300.0	0	0

Table 6.1: Enumerating multiple diagnoses.

MKnapsack : The multi dimensional Knapsack Problem. This involves a large number of overlapping knapsack constraints. The fault added replaces constraints less-equal and greater-equal with constraints less-than and greater-than, respectively.

NMSeq : A naive model for the Magic sequence problem. The injected fault replaces an equality with a disequality: constraint `==` became `!=`.

Open-Stacks : This is a scheduling problem where a sequence of product orders must be found that minimises the number of orders open at the same time. The injected fault is an indexing issue, where a constant array access to the first element of a zero-indexed array is changed: array access `o[i,0]` became `o[i,1]`

P1F : The goal with this problem is to find a perfect 1-factorization of a graph. The added fault replaces a disequality with an equality: constraint `!=` became `==`.

Radiation : The goal here is to find an arrangement of screens with blocked sections so that certain amounts of radiation can be delivered through the screens in a treatment. The added fault uses the wrong index variable for the coefficients used in an expression: expression `b * Q[i,j,b]` became `i * Q[i,j,b]`.

Spot5 : This problem is an earth observation satellite management problem. The fault added offsets an expression by adding 1 to it: expression `constraints2[u]` became `constraints2[u]+1`.

TDTSF : The time dependant travelling sales person problem. The added fault is another off-by-one mistake: index `n+1` became `n`.

6.5.1 MUS enumeration

Table 6.1 shows a comparison of the different configurations when enumerating multiple MUSes. The columns for each configuration show the duration in seconds (T) taken in finding the MUSes for the FlatZinc program obtained for the model's instance, the number of constraint groups (G) explored, and the number of MUSes found (M). Bold text is used to highlight which configuration was faster for a particular model at each *maxDepth*. It can be seen from the number of cases where configurations reach the 300 second time limit that enumerating all MUSes for a problem can be an expensive task. Comparing

Configuration	C_2			$C_{1 \rightarrow 2}$			C_3			$C_{1 \rightarrow 3}$			$C_{Program}$			$C_{1 \rightarrow Program}$		
Model	T	G	M	T	G	M	T	G	M	T	G	M	T	G	M	T	G	M
Costas-Array	2.5	56	1	3.9	54	1	300.0	83	0	4.7	33	1	300.0	83	0	6.0	33	1
CVRP	0.5	34	1	0.3	7	1	0.9	66	1	0.4	7	1	1.3	101	1	0.8	14	1
Free-Pizza	1.6	81	1	0.6	9	1	1.7	82	1	0.7	1	1	9.4	553	1	0.9	1	1
Mapping	0.4	24	1	0.3	2	1	0.6	69	1	0.4	28	1	1.6	254	1	0.6	4	1
MKnapsack	0.5	31	1	0.5	30	1	0.5	31	1	0.5	1	1	0.6	32	1	0.7	1	1
NMSeq	300.0	40	0	300.0	40	0	300.0	40	0	300.0	0	0	300.0	3240	0	300.0	0	0
Open-Stacks	53.9	802	1	52.1	800	1	73.7	841	1	55.4	7	1	300.0	4421	0	73.4	14	1
P1F	3.9	89	1	1.9	11	1	4.2	89	1	2.1	1	1	29.6	947	1	2.4	1	1
Radiation	0.5	5	1	0.5	4	1	1.4	68	1	0.9	25	1	6.5	388	1	1.2	1	1
Spot5	300.0	4998	0	258.9	4406	1	300.0	5227	0	264.2	1	1	300.0	5457	0	264.2	1	1
TDTSP	1.2	46	1	0.5	1	1	1.4	62	1	0.4	1	1	3.0	170	1	0.7	1	1

Table 6.2: Finding the first diagnosis.

configurations C_2 and $C_{1 \rightarrow 2}$, we see that $C_{1 \rightarrow 2}$ is often faster, although in three of the eleven cases it cannot find as many MUSes as the fixed depth C_2 configuration. For example, for **Free-Pizza** configuration C_2 returns 13 MUSes before timing out, while $C_{1 \rightarrow 2}$ can only yield a single MUS, having discarded too many groups.

Comparing configurations C_3 and $C_{1 \rightarrow 3}$, shows that there are also several cases where configuration $C_{1 \rightarrow 3}$ is faster (**CVRP**, **Free-Pizza**, **Mapping**, and **Radiation**), although again C_3 can find more MUSes and, surprisingly, is faster for two models (**MKnapsack**, **P1F**).

Finally, comparing configurations $C_{Program}$ and $C_{1 \rightarrow Program}$, the deepening approach discovers MUSes at the level of individual program level constraints much faster than the full enumeration approach in three cases (**CVRP**, **Free-Pizza**, and **Radiation**). An interesting result can be seen in the **MKnapsack** instance, where increasing the depth does not change the number of constraint groups found. Further, since the enumeration algorithm has to essentially repeat enumeration at each depth before going deeper, it ends up being much slower. This indicates that the deepening approach may be less useful when most of the constraints are in conflict. It also suggests an optimisation that, if implemented, will not allow for fruitless deepening.

Ultimately for enumeration of MUSes, both the selective deepening and fixed-depth approaches are viable, but the selective deepening should probably be attempted first as it is often faster.

6.5.2 Time to First MUS

In practice a user will often start trying to fix the first few MUSes that are presented rather than wait for a full enumeration. This is similar to debugging in traditional programming languages, where one would rarely try to fix multiple reported errors at once but, instead, fix one and then check whether it was responsible for the cascade of remaining errors. With this in mind, a second experiment, presented in Table 6.2, explores how quickly the configurations can find their first MUS at a target depth.

Configurations C_2 and $C_{1 \rightarrow 2}$ in this case show that the incomplete enumeration configuration ($C_{1 \rightarrow 2}$) is almost always faster even at the shallow depth of 2. The fixed-depth approach is only significantly faster in one case (**Costas-Array**). An interesting case at this depth is **Spot5**, where $C_{1 \rightarrow 2}$ can find a MUS in the set of 4406 groups, while the fixed-depth

approach fails to do so in the larger set of 4998. This shows that even small reductions in the number of constraints can have a big impact on performance.

C_3 and $C_{1 \rightarrow 3}$ show an even more pronounced effect, with $C_{1 \rightarrow 3}$ being faster or equal in all cases. The impact of the incomplete enumeration can be seen in the results for the *Free-Pizza* model, where C_2 must find MUSes given 82 groups at depth 2, whereas configuration $C_{1 \rightarrow 2}$ has narrowed the set of constraints to a single constraint group. An issue is highlighted by the performance of the approach on the *NMSeq* model. The table shows that $C_{1 \rightarrow 3}$ is exploring a set of zero constraint groups. This is because, after 300 seconds, the algorithm had not reached the target depth and was still searching in the set of 40 constraints at a depth of 2. This suggests that the fixed-depth approach might actually be more suitable in certain cases. However, predicting this is challenging.

Finally, with $C_{Program}$ and $C_{1 \rightarrow Program}$, the traditional program-level approach for finding a single MUS is compared against the incomplete deepening approach. Again the incomplete deepening approach outperforms $C_{Program}$ in almost all cases by honing in on a single MUS rather than getting lost searching in the full set of program-level constraints. The results for the *Open-Stacks* problem demonstrate that the incomplete deepening approach can outperform the traditional approach by quite a large margin, with $C_{Program}$ having to search through the powerset of 4421 constraints, while $C_{1 \rightarrow Program}$ has narrowed this down to a set of only 14 constraints. This huge difference in performance shows that the incomplete deepening approach can be effective for discovering MUSes.

6.6 Displaying Diagnoses

The approach presented in Section 6.4 reports diagnoses as sets of MiniZinc paths. While paths contain all the information required to find the conflicting constraints, they are not easy to read by humans. One approach is to require users to explicitly group related constraints together under user defined predicates, mimicking the approach taken in Jussien et al. (2001). The predicate names will then be included in the constraint paths, making the paths a little easier to follow. However, even with these hints the paths might still be difficult to read. Additionally, users may be reluctant to go through the tedious process of picking through the paths.

To improve this situation, diagnoses must be presented in a more useful form than a set of constraint paths. The paths encode the set of syntactic positions that a diagnosis relates to, and the specific combination of assignments to loop index variables during compilation that make up the diagnosis. An extension to the MiniZinc IDE has been developed which can interpret and display this information directly in a source code editor.

Case Study: Over-constrained Latin Squares

Section 6.2 introduced a faulty model for the Latin Squares problem. Figure 6.6 shows the model in the MiniZinc IDE editor. Using the techniques presented in this chapter on this model instantiated with $n = 3$, exposes several MUSes. Figure 6.7 shows how the IDE presents these to the user. Each MUS is displayed in the output section at the

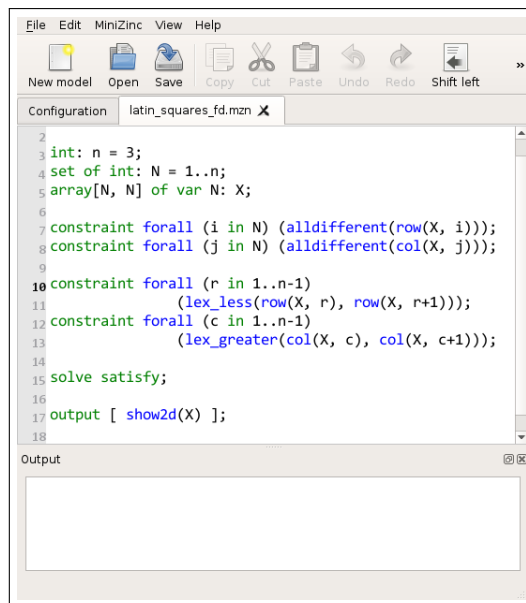


Figure 6.6: Over-constrained Latin Squares in the MiniZinc IDE.

bottom, with the number of constraint groups involved and the list of assignments to index variables that lead to each group being introduced.

The highlighting in the model presented in Figure 6.7a marks the constraints involved in the selected MUS. Note that all MUSes for this problem involve some combination of the `alldifferent` constraints and the `lex` constraints. However, looking at the *intersection of constraint groups* involved in all MUSes, it is easy to find that they all include at least 3 of the `lex` constraints. It follows that these `lex` constraints are likely candidates for being the real source of unsatisfiability. With this hint, the user can re-examine the reasoning behind the inclusion of these constraints and, hopefully, understand the conflict.

If users cannot immediately deduce the cause of unsatisfiability, they can look at which specific rows and columns are involved by looking at the assignments that are listed with each MUS. For example, the MUS `Conflict:5: c=2; i=1; i=2; c=1; r=1` highlighted in Figure 6.7a, involves 5 constraint groups, and indicates that the fault involves both of the `lex_greater` constraints over the first three columns (`c=1; c=2`), the `alldifferent` constraints on the first two rows (`i=1; i=2`), and the `lex_less` over the first two rows `r=1`.

If the enumeration algorithm is configured to find MUSes at greater depths, the MUSes will include constraints introduced by decompositions of constraints. This can be seen in Figure 6.7d where a user has clicked on one such MUS. Tabs for displaying the place in the decompositions of the `lex_less_int` and `all_different_int` constraints are opened automatically. The highlighting shows a specific **less-than-equal** constraint that is involved in this MUS. Tracking down constraints across multiple MiniZinc files is made much easier by this feature. This would mostly be useful for experts users, since new users will probably only want to focus on the faults in their own models, rather than in the decompositions.

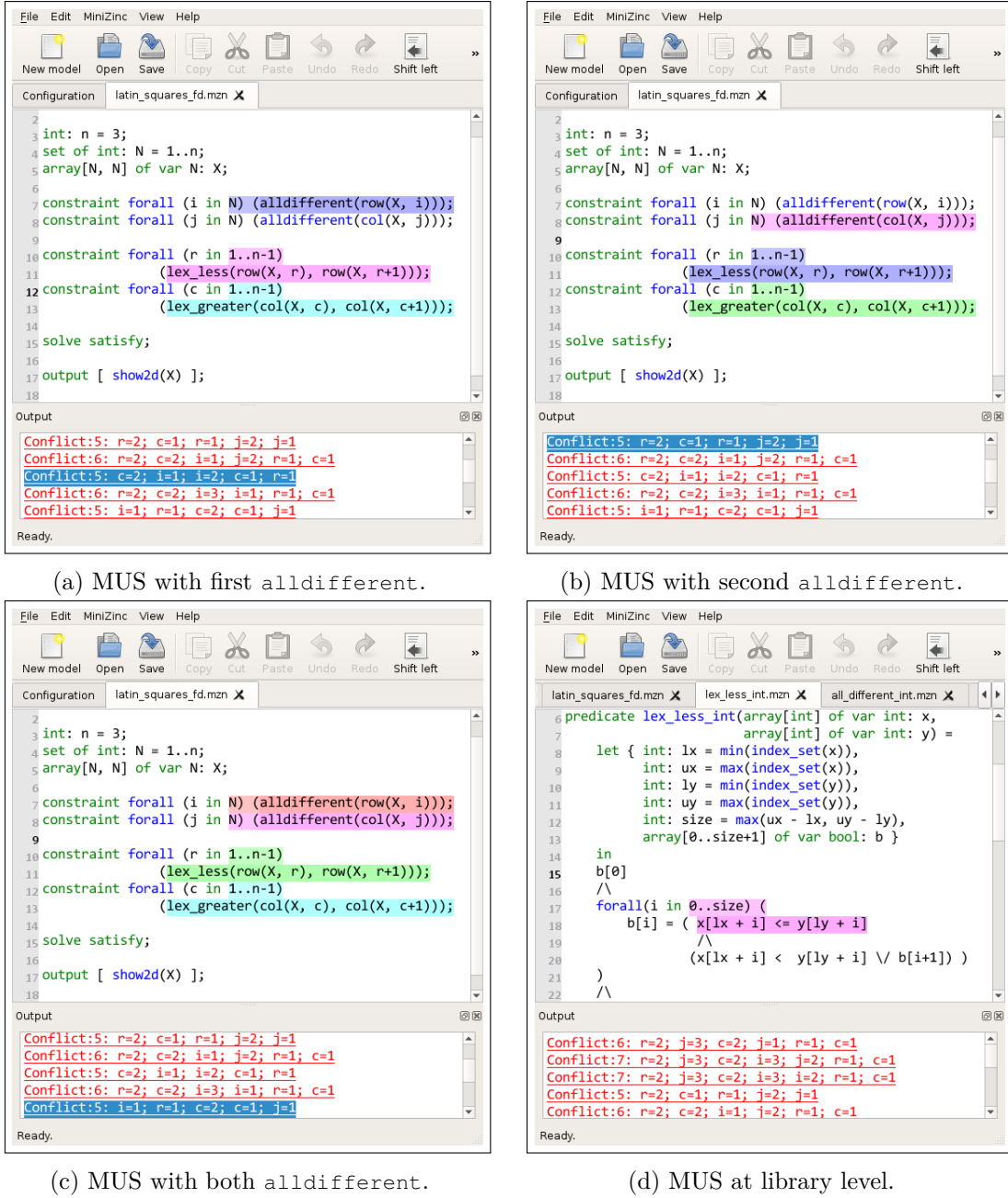


Figure 6.7: Prototype MiniZinc IDE showing different diagnoses.

6.7 Generalising MUSes to the Model Level

MiniZinc paths were introduced to identify common structures across multiple compilations of the same instance. With a slight modification, they can also provide insights into the common structures shared between different instances, generalising these structures by expressing them in terms of the parametric model.

In some cases multiple instances of a problem will have some common MiniZinc paths. For example, paths for the initial iterations of a `forall` loop will usually be the same. In the Latin Square model presented in Section 6.2, all valid values for the parameter `n` result in the first `forall` being evaluated for `i = 1`. Different instances will have similar

paths for these constraints. Since the instances are different, the constraints cannot be considered to be the same and, thus, no automatic reasoning on these can be performed. A similar approach to that presented in Section 6.6 can be applied here. We previously discussed how users could look at the intersection of MUSes to find the constraints that occur in most MUSes. Here, this idea is extended and used to explore the intersection of model-level (parametric) MUSes instead. However, in a similar vein to Chapter 5 these MUSes can only be provided to the user as *suggested* model-level structure. It is up to the user to determine whether the suggested structure holds or not. These MUSes are parametric and can only say that a set of parameters exist that make them unsatisfiable.

To demonstrate this, we will return to the Latin Squares example from Section 6.2. Trying to compare MUSes found for an instance with $m = 4$ against MUSes found for $m = 12$, we find that many MUSes cannot be directly compared as they involve rows or columns that do not exist in the smaller instance. However, the particular combination of constraints that cause unsatisfiability is the same. Thus, we need to find a way to compare MUSes when the instances are different. This is achieved by *generalising* the paths of the constraints in the MUSes by replacing the instance dependant parts of each path with placeholders before comparing, e.g., by replacing the loop index assignment $i=1$ with $i=\$$. This way, all iterations of a loop will be grouped together when grouping by path. While this means that we can no longer mark the specific iteration that is faulty, we still know there is at least one faulty iteration in every instance. Importantly, these generalised paths still retain enough of the explicit structure from the model for the selective deepening approach to be applied. A more nuanced form of generalisation could be implemented if this is not specific enough. For example, place-holders signifying whether a constraint comes from the first half of the iterations of a loop could be used, i.e., $i=4$ becoming $i=LEFT$ or $i=RIGHT$ depending on the range of i .

Grouping instances by their generalised MUSes: Generalised MUSes are particularly useful when some of the set of instances should be satisfiable, while others should be unsatisfiable (assuming a correct model). If it is not known which instances should be unsatisfiable, the process of debugging a model can be difficult, as the user will have to deduce whether unsatisfiability is due to a bug in the model or the instance data. Using the techniques presented in this chapter, the conflicts arising from each set of instances can quickly be discovered and compared, giving the user a better idea of what may be happening with their model. In particular, just as the intersection of MUSes from a single instance was used to find which constraints are common to a set of MUSes, the intersection of generalised MUSes can be used to group the instances. To do this, generalised MUSes are presented to the user ranked by the number of instances that they occur in. Looking at this ranked list, a user can discern which generalised MUSes are common to all instances, indicating potential modelling bugs, and which generalised MUSes only occur in specific instances, indicating unsatisfiable instances. A proof of concept implementation of this approach was developed. It takes the form of a small Python script that takes a MiniZinc model and a set of data files as input, along with parameters for the *diagnose* procedure. The following explores a case where this implementation of the approach proves useful.

Case Study: RCMSP

To demonstrate how this approach can work in practice, a complex model along with a set of satisfiable and unsatisfiable instances is explored here. The selected model implements the Resource-Constrained Modulo Scheduling Problem (RCMSP) that was explored in Chapter 4, and presented as Listing 4.11. A simplified version of this model was presented in Section 4.5.3.

```

37 constraint forall(r in Res) (
38   let {
39     set of int: ResTasks =
40       { i | i in Tasks where rreq[i, r] > 0 /\ d[i] > 0 },
41     int: sum_rreq = sum([0] ++ [rreq[i, r] | i in ResTasks])
42   } in (
43     if sum_rreq <= rcap[r]
44     then true
45     else cumulative(
46       [ d[i]          | i in ResTasks ],
47       [ s[i]          | i in ResTasks ],
48       [ rreq[i, r]    | i in ResTasks ],
49       rcap[r])
50   endif));

```

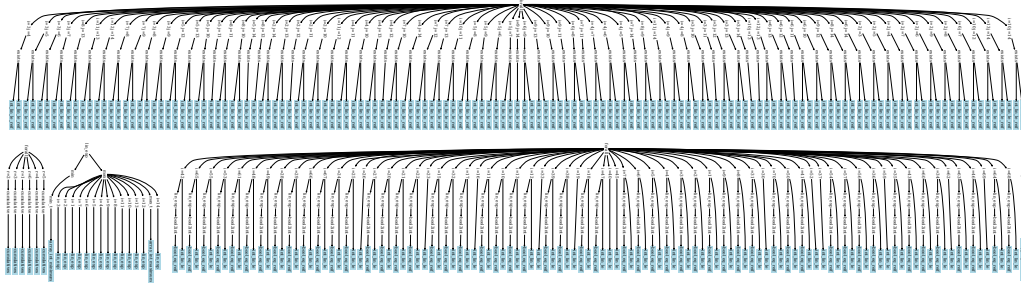
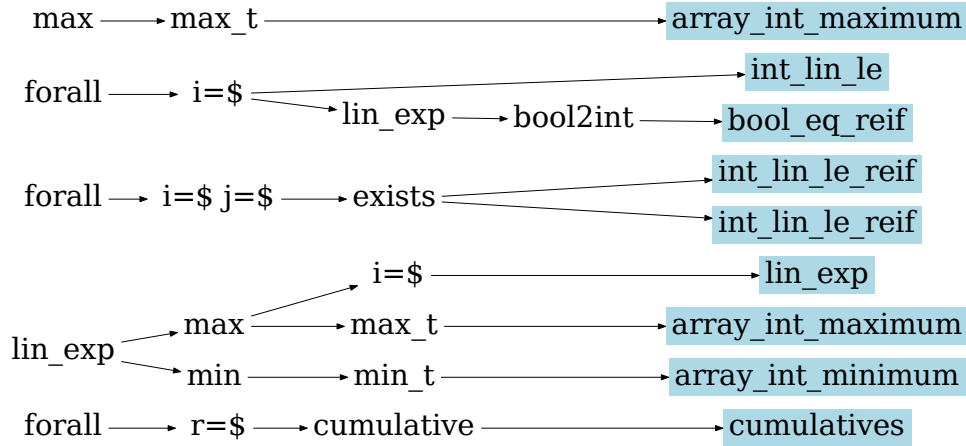
Listing 6.6: Incorrect argument order in call to `cumulative`.

Listing 6.6 shows an extract from the model that introduces several `cumulative` constraints. Note that a fault has been introduced in this model by modifying the call to `cumulative` on lines 45-49, swapping the arguments `s` and `d`, the start time and duration variables. Since both `s` and `d` have the same type signature (`array[int] of int`) the MiniZinc compiler does not detect this mistake.

The instances used include three satisfiable instances, along with two pairs of unsatisfiable instances: two instances with unsatisfiable resource capacities; and two with cyclical task precedences (task *A* depends on task *B*, which in turn depends on task *A*).

Figure 6.8 presents a trace of the compilation of one of the instances of the model. It is clear from this figure that the programs for these instances can get large, making MUS enumeration a challenging and time consuming task even for just a single instance. We are interested, however, in finding generalised MUSes for several instances. Applying the generalisation approach outlined above, which replaces identifying information with placeholders, results in the collapsed tree shown in Figure 6.9. Note that there are no longer any program-level constraints in the tree, and the leaves are in fact groups of constraints with the same paths.

This grouping means that the MUS enumeration algorithm only needs to look through combinations of at most 9 groups regardless of the instance data, as the data can only change which groups trigger unsatisfiability. MUSes for each instance can now be discovered quickly. The output from the Python script that automates this process is presented in Listing 6.7. Five distinct generalised MUSes can be seen here, occurring in instances $\{0,1,2,5,6\}$, $\{3,4\}$, $\{1,5\}$, $\{2,6\}$ and $\{2\}$, respectively. Since the first generalised MUS occurs in 6 of the 7 instances, it is a strong candidate for being a model-level fault.

Figure 6.8: Full trace tree for RCMSP instance `medium_2.dzn`.Figure 6.9: Resulting *generalised* tree for the RCMSP problem. MUSes found in this tree can be compared across instances.

Indeed, this MUS involves the incorrect arguments to `cumulative`. The instance that does not include this exact MUS (4) also relates to `cumulative` but it fails in a slightly different way. Once the bug is fixed, the analysis can be executed again. The output of this is shown in Listing 6.8. The remaining MUSes occur in instances {3,4} and {2,6} which correspond to the instances with the two different injected faults. This shows that generalised MUS enumeration is a useful and powerful approach that can make debugging unsatisfiable models and instances considerably easier.

MUS: Instances

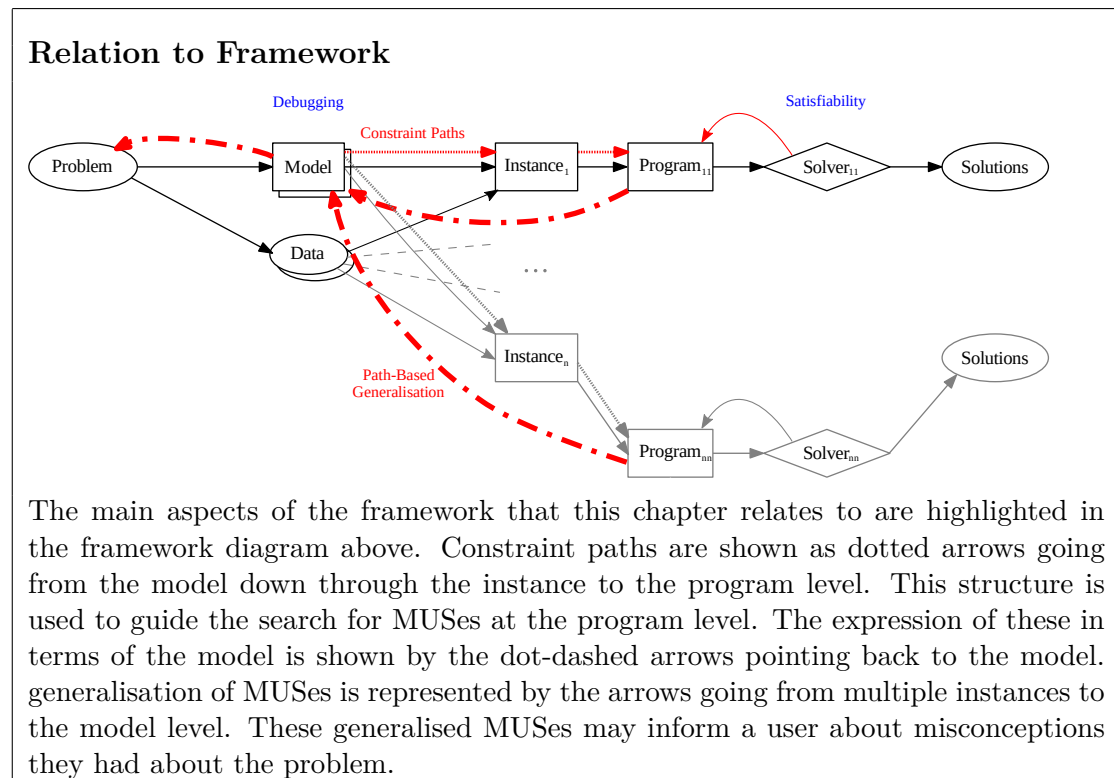
```
0: 0 1 2 5 6
1: 3 4
2: 1 5
3: 2 6
4: 2
```

Listing 6.7: MUSes occurring in several instances.

MUS: Instances

```
1: 3 4
2: 2 6
```

Listing 6.8: MUSes remaining after correcting bug.



6.8 Conclusion

When faced with debugging an unsatisfiable model, users are typically on their own. Generating meaningful diagnoses can make the task easier. The main contribution of this chapter is an approach for finding Minimal Unsatisfiable Subsets (MUSes) in constraint models that is more scalable and presents the MUSes in such a way that allows users to quickly find the source of unsatisfiability.

Explicit model-level structure is utilised to automatically extract the constraint hierarchy using the novel concept of constraint paths, and to group related constraints together during MUS enumeration. This reduces the search space and can speed up the discovery of diagnoses. Using constraint paths, the exact pieces of code that are unsatisfiable when combined, can be highlighted in the user’s model.

Further, a methodology is presented for deducing whether the model is faulty, given a set of satisfiable and unsatisfiable instances. This approach also allows unsatisfiable instances to be classified by showing the generalised MUSes they have in common.

6.8.1 Limitations.

The approach presented in this chapter makes use of the explicit hierarchy present in a model of a constraint problem. While our experimental evaluation has shown that this can be an effective choice in practice, it is possible to conceive of pathological cases where this does not work well. For example, examining Figure 6.8 we see that once we leave the root-level of the hierarchy, the frontier is expanded into a large number of constraints that

must be explored. As such, it may make sense to implement some other form of binary splitting that introduces extra nodes that split the constraints below in half.

As the area of MUS enumeration research has been focused on finding program-level MUSes, there is no existing corpus of unsatisfiable models that can be used to effectively benchmark our approach. The collection of unsatisfiable models from users who are new to modelling constraint problems, is a time consuming and labour intensive task that is considered beyond the scope of this thesis but may be worth pursuing in future.

Similarly, we do not attempt to test the effectiveness of the user interface. While, intuitively, it should be useful, we can only offer anecdotal evidence for now. User studies are out of scope of this thesis, since the focus of the work was on improving MUS enumeration.

6.8.2 Further Research.

The approaches explored here will be integrated into the MiniZinc compiler. This will provide a more consistent interface for users than the prototype system, which involves running a Python script to find the MUSes and load them into the IDE for examination.

The UI for displaying MUSes in the MiniZinc IDE, while already useful, could be improved by showing the values of index variables in the modelling window, making it easier for users to figure out which specific iterations are involved. Further, as mentioned above, we have not focused on the usability of the user interface. The application of a user focussed design methodology to develop a better interface for the debugging tool would be interesting, and beneficial for the community.

The speed of using incomplete enumeration to find a single MUS also suggests a further optimisation for the enumeration of MUSes. Enumeration is currently implemented as a breadth-first search, finding all MUSes at a certain depth before deepening. An improvement to this would be to use the incomplete enumeration to find a single MUS in a backtracking depth-first manner to enumerate MUSes.

Finally, using a solver based on Lazy Clause Generation (Feydy et al., 2009; Ohrimenko et al., 2007) to maintain the *map* in the MARCO algorithm may offer a more expressive language for finding unexplored subsets and intelligently blocking subsets that do not need to be checked.

Chapter 7

Conclusions

High-level modelling languages allow combinatorial problems to be specified as high-level models that can then be compiled into different low-level programs, suitable for being solved by different solvers. While the high-level modelling and solver-independence of these languages is useful for modellers, they make it difficult for the compiler to yield programs that can be solved efficiently. This makes the modelling, compilation, and solving of combinatorial problems a challenging iterative process.

This thesis explores in detail the process and presents improvements to both the modelling and compilation stages. These improvements have made the entire process more efficient and, as a result, more attractive to users. One of the key weaknesses identified in the process was the fact that much of the explicit structure present in a model was not being utilised to its full potential. The improvements presented in this thesis all focus on the goal of *taking advantage of the explicit and implicit structure in constraint models to improve the modelling, compilation, and solving process*. This chapter presents the conclusions to the thesis focusing on its contributions, and discusses the future work arising from them.

A framework for model structure. The first main contribution of this thesis, presented in Chapter 3, is a framework for reasoning about the flow of structural information within the modelling, compilation, and solving process. The framework presents a more complete view of the process that includes the multiple programs that can result from an instance, and the multiple instances that can result from the same model. By doing this, the framework exposes opportunities for discovering inefficiencies and waste in the process. The main contributions of the new framework are as follows:

- It shows many new routes the flow of structural information can take within the modelling, compilation, and solving of a problem; routes that are omitted from the traditional model.
- It highlights the importance of good models by showing how the quality of structural information available at each successive stage in the process, is derived from the model, and is thus critically dependant on its quality.

- It allows the identification of gaps in the existing process, where structural information is not being adequately discovered and used. This puts in context the methods introduced in this thesis and suggests further improvements that could be made.
- It shows how structural information can be both shared within a stage and generalised to earlier stages, to improve the entire process.

Of course, the framework is not complete and there is room for further research. First, the thesis only explores a few improved uses of information sharing and generalisation within the framework, covering only a subset of the possible improvements that could be made. And second, more components of the framework could be introduced to strengthen the links in the process. For example, the framework does not address the idea that multiple models for a problem can be used in conjunction with each other. While the iterative process of improving a model is shown (as a red self-arrow on a stack of models), it could be expanded further. The ESSENCE (Gent et al., 2007b) modelling language explores the idea that, from an abstract description of a problem, many, still high-level models can be automatically specified. There is definitely room for exploring the sharing of information between these models (e.g., sharing of information between different models solving the same instance).

Preserving structure. The second main contribution of this thesis, presented in Chapter 4, is a method to preserve the explicit structure of a model during the compilation process, and to use this structure to improve the quality of the resulting compiled program. This approach demonstrated that *explicit structure in models can be used to improve the quality of programs compiled from them*. The main contributions of this work are as follows:

- It introduces the novel concept of variable paths, that is, unique variable identifiers that are stable across compilations and can, thus, be used in the matching of structural information in multiple compilations of an instance of a model.
- It shows how the sharing of information within the modelling, compilation, and solving process is much easier, thanks to the introduction of variable paths.
- In particular, it shows how for the first time the compiler can fully combine the strengths of multiple solvers in producing efficient programs, that is, how the compiler can communicate information inferred by different solvers not just on top level variables but also, on those introduced by the compiler. This has never been possible before and significantly increases the possibilities for improving the resulting programs.
- It narrows the gap between hand-written programs and those produced by compilers for high-level modelling languages, making the use of these languages much more attractive for both experts and new users.
- It implements the approach for the MiniZinc tool chain, and a release is planned.

With the multi-pass architecture in place in the MiniZinc compiler, there is now some room for exploring the integration of new kinds of passes that can improve the compilation of models, other than the root node propagation pass. In addition, the variable paths concept could also be utilised in more areas. An interesting application could be the use of paths to provide more useful feedback to users when they use tools such as the CP Profiler (Shishmarev et al., 2016), where paths could be used to tie the nogoods learned by a learning solver back to the user’s model, making it easier for users to explore the solver behaviour. Another application of variable paths could be in allowing more powerful hybrid solvers to be automatically derived for a given model; hybrids that can communicate much more information than was previously possible. A limitation of the multi-pass compilation approach is that it can be difficult to predict when it should be used. Using techniques from machine learning to decide whether it should be used, and if so, how it should be configured for different classes of problem, would also be an interesting research direction.

Discovering structure. The third main contribution of this thesis, presented in Chapter 5, is a method for *globalizing* constraint models, that is, for proposing global constraints that can replace parts of a model. It does this by using the *explicit structure of a model to guide the search for and exposure of implicit structure*. This helps users develop clearer models that can be compiled into more efficient programs. The main contributions of this work are as follows:

- It shows how both the model and the data can be used to increase efficiency and accuracy. In particular, it shows how to split the constraints of the model and unroll its loops to increase the number of global constraints considered; how to use the variables, parameters, and expressions of a model to efficiently generate the arguments for the candidate global constraints; and how to use different instances of the same model to remove spurious candidates.
- It captures other known kinds of structure – such as the alternative viewpoints common in dual models – since they can also be expressed in terms of parameterised patterns similar to global constraints. This, when combined with an iterative application of the method, can be used to significantly increase the usefulness of the method.
- It makes modelling easier for users, as it allows them to first focus on designing a correct and intuitive model. Once this is achieved, the system guides them through possible improvements that could be made.
- Again, this makes high-level modelling more attractive to new users who, to get started with the new language, do not have to completely change the way they think about modelling problems before they can start solving them.
- It implements the method in the MiniZinc Globalizer, which was previously available as an online tool. A new version will soon be released and integrated with the

MiniZinc IDE where it can be more easily integrated into a users' existing workflow. A release is planned.

One limitation of the approach is that beyond using the model structure to split a model and construct arguments for global constraints, the search does not use any syntactic matching, and thus, in some pathological cases, can be inefficient. Developing heuristics that can guide the search for globals (e.g., integrating syntactic matching techniques) could allow the approach to become much more robust.

Structure guided fault diagnosis. The fourth and final main contribution of the thesis, presented in Chapter 6, is an automatic approach to the debugging of unsatisfiable constraint problem models that is both scalable and useful. It achieves this by using the *explicit structure of a model to help identify parts of a model that are incorrectly formulated*. The main contributions of this work are as follows:

- It extends the concept of variable paths by defining constraint paths, that is, unique constraint identifiers that are stable across compilations and can, thus, be used in the matching of structural information in multiple compilations of an instance of a model.
- It shows how to use the constraint hierarchy automatically exposed by constraint paths to speed up the search for minimal unsatisfiable subsets. This makes the method scalable.
- It shows how to present program-level MUSes to modellers in a meaningful way, by making use of constraint paths to link the program constraints present in a MUS to the model constraints. This significantly increases the method's usefulness.
- It shows how to generalise MUSes in such a way as to be able to distinguish between genuine modelling bugs from the unsatisfiability that arises from faulty instance data. This is achieved by further generalising the concept of constraint paths – and thus MUSes – so they can be examined across multiple instances of a problem.
- It provides users with more tools to aid their development of a correct version of a model. This again, contributes to the benefits that high-level modelling languages can provide to users.
- A prototype was developed for MiniZinc by modifying an existing MUS enumeration tool. The ability to explore the MUSes of a model in the MiniZinc IDE was also implemented, along with a Python script that automated the exploration of cross-instance MUSes. A native implementation in the MiniZinc compiler is planned for release.

One limitation of the approach is the fact that unsatisfiability is not the only type of fault that can occur in an incorrect model. For example, sometimes a model is satisfiable but returns incorrect solutions. It would be interesting to see whether this approach can

be extended to explain other kinds of model faults. In addition, other extensions of the constraint and variable paths for use in the debugging or profiling of constraint solvers could also be explored. For example, paths could make the comparison of search trees from two different solvers operating on different programs much easier and could give users insight into their models.

Summary. In conclusion, this thesis presents a methodology for exploring the flow of structural information in the modelling, compilation, and solving process, and how it can be improved. In addition, it presented three other significant contributions that, together, make high-level modelling a more powerful and attractive tool for those who need to solve hard combinatorial problems.

Appendix A

Preserving Strucutre: Models

The models and instances used in the experimental evaluation of the path based multi-pass presolving presented in Chapter 4 were downloaded from the results pages for the MiniZinc challenge, years 2012, 2013, and 2014 (<http://www.minizinc.org/challenge<year>/results<year>.html>). Each model comes with 5 data files. The selected problems from Figure 4.4 are repeated below.

Amaze-2	Amaze-3	Black-hole	Cargo-coarse
Carpet-Cutting	Celar	Fastfood	Fillomino
Filter	FJSP	FM	Ghoulomb
Handball	Jp-encoding	League	Linear2program
Mario	Mknapsack	Mqueens	MSPSP
NMSeq	Nonograms	OC-Roster	Openshop
Parity-learning	Patternset-mining	Pentominoes-int	ProjectPlanner
Radiation	RCMSP	RCPSP	Rect-packing
Road-Cons	Rubik	SB	Ship-Schedule
Smelt	Spot5	Still-life	TP
TPP	TPPPV	Train	Trip
VRP	WCSP		

These models, along with many others, and their data files can also be acquired from the MiniZinc benchmarks repository which can be found at <https://github.com/MiniZinc/minizinc-benchmarks>.

Appendix B

Discovering Structure: Models

Adapted versions of the models used for the experimental evaluation of the Globalizer approach presented in Chapter 5 are reproduced here.

B.1 Cars

A model for this problem was presented in Listing 1.1.

B.2 Jobshop

```

1  int: size = 2;
2  array [1..size,1..size] of int: d;
3  int: total = sum(i,j in 1..size) (d[i,j]);
4
5  array [1..size,1..size] of var 0..total: s;
6  var 0..total: end;
7
8  constraint end != end;
9
10 constraint
11   forall(machine in 1..size) (
12     forall(j in 1..size-1)
13       (s[machine,j] + d[machine,j] <= s[machine,j+1]) /\
14       s[machine,size] + d[machine,size] <= end /\
15       forall(j,k in 1..size where j < k) (
16         s[j,machine]+d[j,machine] <= s[k,machine]   \/
17         s[k,machine]+d[k,machine] <= s[j,machine])
18   );
19
20 solve minimize end;
```

B.3 Party

This model was presented in Listing 5.1.

B.4 Party-CSP

```

1  include "all_different.mzn";
2
3  int: p; int: nh; int: ng;
4  set of int : HostBoats = 1..nh;
5  set of int : GuestCrews = 1..ng;
6  set of int : Time = 1..p;
7
8  array [GuestCrews] of int : crew;
9  array [HostBoats] of int : capacity;
10
11 array [GuestCrews, Time] of var HostBoats : hostedBy;
12
13 array [GuestCrews, HostBoats, Time] of var 0..1 : visits;
14
15 constraint forall (i in GuestCrews)
16   (alldifferent([h[i,t] | t in Time]));
17
18 constraint forall (t in Time, i in GuestCrews, j in HostBoats)
19   (visits[i,j,t] = 1 <-> hostedBy[i,t]=j);
20
21 constraint forall (t in Time, j in HostBoats)
22   (sum (i in GuestCrews) (crew[i]*visits[i,j,t]) <= capacity[j]);
23
24 array [GuestCrews, GuestCrews, Time] of var 0..1 : meet;
25
26 constraint forall (k in GuestCrews, l in GuestCrews, t in Time where k<l)
27   (hostedBy[k,t] = h[l,t] -> meet[k,l,t] = 1);
28
29 constraint forall (k in GuestCrews, l in GuestCrews where k<l)
30   (sum (t in Time) (m[k,l,t]) <= 1);
31
32 solve satisfy;

```

B.5 Party-LP

This model was presented in Listing 5.8.

B.6 Packing

```

1  int: pack_x;
2  int: pack_y;
3  int: n;
4  array[1..n] of int: pack_s;
5
6  array[1..n] of var 0..pack_x-1: x;
7  array[1..n] of var 0..pack_y-1: y;
8
9  constraint
10     forall (i in 1..n) (
11         x[i] <= pack_x - pack_s[i] /\
12         y[i] <= pack_y - pack_s[i]
13     );
14
15  constraint
16     forall (i in 1..n-1, j in i+1..n) (
17         x[j] - x[i] >= pack_s[i] /\
18         x[i] - x[j] >= pack_s[j] /\
19         y[j] - y[i] >= pack_s[i] /\
20         y[i] - y[j] >= pack_s[j]
21     );
22
23  constraint
24     forall (i in 1..n-1) (
25         if pack_s[i]=pack_s[i+1] then
26             x[i] <= x[i+1]
27         else
28             true
29         endif
30     );
31
32  constraint
33     forall (cx in 0..pack_x-1) (
34         sum (i in 1..n)
35             (pack_s[i]*bool2int(x[i] in cx-pack_s[i]+1..cx)) = pack_y
36     );
37  constraint
38     forall (cy in 0..pack_y-1) (
39         sum (i in 1..n)
40             (pack_s[i]*bool2int(y[i] in cy-pack_s[i]+1..cy)) = pack_x
41     );
42
43  solve :: seq_search([int_search(x,smallest,indomain_min,complete),
44                     int_search(y,smallest,indomain_min,complete)])
45      satisfy;

```

B.7 Schedule

```

1  int: d;
2  int: n;
3  array[1..n] of int: m;
4  int: maxR;
5  int: horizon = 2*(n-1);
6  int: makespan = horizon+d;
7
8  array[1..n] of var 0..2*(n-1): x;
9  array[0..2*(n-1), 1..n] of var 0..maxR: R;
10
11 constraint forall (i,j in 1..n where i<j)
12     ((x[i]-x[j] >= 2) /\ (x[j]-x[i] >= 2));
13
14 constraint forall (t in 0..horizon, i in 1..n) (
15     let {
16         var bool: active = (x[i] <= t /\ t<x[i]+d)
17     } in
18     ( active <-> R[t,i] = m[i] ) /\
19     ( (not active) <-> R[t,i] = 0 )
20 );
21
22 constraint forall (t in 0..horizon) (sum (i in 1..n) (R[t,i]) <= maxR);
23
24 solve satisfy;

```

B.8 Sudoku LP

```

1  int: S = 3;
2  int: N = S*S;
3  array[1..N, 1..N] of int: givens;
4
5  array[1..N, 1..N, 1..N] of var 0..1: x;
6
7  constraint forall(i,j in 1..N where
8      givens[i,j] != 0) (
9      x[i,j, givens[i,j]] = 1
10 );
11
12 constraint forall(i in 1..9, j in 1..9) (sum([x[i,j,k] | k in 1..9]) = 1);
13 constraint forall(i in 1..9, k in 1..9) (sum([x[i,j,k] | j in 1..9]) = 1);
14 constraint forall(j in 1..9, k in 1..9) (sum([x[i,j,k] | i in 1..9]) = 1);
15 constraint forall(I,J in 0..2,
16     k in 1..9) ( sum([x[i,j,k] | i in (I*3+1)..(I*3+3),
17     j in (J*3+1)..(J*3+3)]) = 1);
18
19 solve satisfy;

```

B.9 Sudoku CSP

```
1  int: S = 3;
2  int: N = S*S;
3  array[1..N, 1..N] of int: givens;
4  array[1..N, 1..N] of var 1..9: p;
5
6  constraint forall(i in 1..N, j in 1..N) (
7      if givens[i,j] != 0 then p[i,j] = givens[i,j] else true endif
8  );
9
10 constraint forall(r1 in 1..N, c1 in 1..N, r2 in 1..N, c2 in 1..N) (
11     if ((not (r1 = r2 /\ c1 = c2)) /\
12         ((r1 = r2)
13          \/ (c1 = c2)
14          \/ ( ((r1-1) div S = (r2-1) div S)
15              /\ ((c1-1) div S = (c2-1) div S))))
16     then
17         p[r1,c1] != p[r2,c2]
18     else
19         true
20     endif
21 );
22
23 solve satisfy;
```


B.10 Warehouses

```

1  include "globals.mzn";
2
3  int: n_suppliers;
4  int: n_stores;
5  int: building_cost;
6  array[1..n_suppliers] of int: capacity;
7  array[1..n_stores,1..n_suppliers] of int: cost_matrix;
8  int: MaxCost;
9  int: MaxTotal;
10
11 array[1..n_stores] of var 1..n_suppliers: supplier;
12 array[1..n_suppliers] of var 0..1: open;
13 array[1..n_stores] of var 1..MaxCost: cost;
14 var 1..MaxTotal: total;
15
16 constraint
17   sum (i in 1..n_suppliers) (building_cost * (open[i])) +
18   sum (i in 1..n_stores) (cost[i])
19   = total;
20
21 constraint
22   forall (i in 1..n_stores) (
23     cost_matrix[i,supplier[i]] = cost[i]
24   );
25
26 array [1..n_suppliers] of var 0..ub_array(capacity) : use;
27
28 constraint
29   forall (i in 1..n_suppliers) (
30     count(supplier,i,use[i]) /\ use[i] <= capacity[i]
31   );
32
33 constraint
34   forall (i in 1..n_suppliers) (
35     (exists (j in 1..n_stores) (supplier[j] == i)) == (open[i]=1)
36   );
37
38 solve
39   :: int_search(
40     supplier ++ cost ++ [(open[i]) | i in 1..n_suppliers],
41     first_fail,
42     indomain_split,
43     complete
44   )
45   minimize total;

```

Appendix C

Structure Guided Fault Diagnosis: Models

This section presents models which have had faults added to them to make them unsatisfiable for use in the evaluation of the structure guided fault diagnosis techniques presented in Section 6.7. These are adapted somewhat for brevity. The original, non-faulty versions of these can be found at <https://github.com/MiniZinc/minizinc-benchmarks>. The models are available under the MIT licence.

C.1 Costas-Array

```

1  include "alldifferent.mzn";
2
3  int: n;
4
5  array[1..n] of var 1..n: costas;
6  array[1..n,1..n] of var -n+1..n-1: differences;
7
8  constraint alldifferent(costas);
9
10 constraint
11     forall(i,j in 1..n) (
12         if i < j then
13             differences[i,j] = costas[j] - costas[j-i]
14         else
15             differences[i,j] = 0
16         endif
17     );
18
19 constraint
20     forall(i in 1..n-1) (
21         alldifferent(j in 1..n where i < j) (differences[i,j])
22     );
23
24 constraint
25     symmetry_breaking_constraint(

```

```

26         if 1 < n then costas[1] < costas[n] else true endif
27     );
28
29 constraint
30     redundant_constraint(
31         forall(i,j in 1..n where i < j)(
32             differences[i,j] != 0
33         )
34     );
35
36 % Fault added: differences[k-2,l-1] became differences[k-2,l-2]
37 constraint
38     redundant_constraint(
39         forall(k,l in 3..n where k < l)(
40             differences[k-2,l-2] + differences[k,l] =
41             differences[k-1,l-1] + differences[k-1,l]
42         )
43     );
44
45 solve :: int_search(costas, input_order, indomain_min, complete)
46     satisfy;

```

C.2 CVRP

```

1  include "circuit.mzn";
2
3  int: N;
4  int: Capacity;
5  int: nbVehicles = N;
6  int: nbCustomers = N;
7  int: timeBudget = sum (i in 1..N) (max([ Distance[i,j] | j in 1..N]) );
8  set of int: VEHICLE = 1..nbVehicles;
9  set of int: CUSTOMER = 1..nbCustomers;
10 set of int: TIME = 0..timeBudget;
11 set of int: LOAD = 0..Capacity;
12 set of int: NODES = 1..nbCustomers+2*nbVehicles;
13 set of int: DEPOT_NODES = nbCustomers+1..nbCustomers+2*nbVehicles;
14 set of int: START_DEPOT_NODES = nbCustomers+1..nbCustomers+nbVehicles;
15 set of int: END_DEPOT_NODES =
16     nbCustomers+nbVehicles+1..nbCustomers+2*nbVehicles;
17 array[1..N] of int: Demand;
18 array[NODES] of int: demand = [
19     if i <= N then Demand[i] else 0 endif
20 | i in NODES];
21 array[1..N+1, 1..N+1] of int: Distance;
22 array[NODES, NODES] of int: distance = array2d(NODES,NODES,[
23     if i<=nbCustomers /\ j <= nbCustomers then
24         Distance[i+1,j+1]
25     elseif i<=nbCustomers /\ j>nbCustomers then
26         Distance[1,i+1]
27     elseif j<=nbCustomers /\ i>nbCustomers then

```

```

28     Distance[j+1,1]
29 else
30     Distance[1,1]
31 endif
32 | i,j in NODES ]);
33
34 array[NODES] of var NODES: successor;
35 array[NODES] of var NODES: predecessor;
36 array[NODES] of var VEHICLE: vehicle;
37 array[NODES] of var LOAD: load;
38 array[NODES] of var TIME: arrivalTime;
39 var 0..timeBudget: objective;
40
41 constraint redundant_constraint(
42     forall(n in (nbCustomers+2..nbCustomers+nbVehicles)) (
43         predecessor[n] = n + nbVehicles-1
44     )
45 );
46
47 constraint redundant_constraint(
48     predecessor[nbCustomers+1] = nbCustomers+2*nbVehicles
49 );
50
51 constraint
52     forall(n in (nbCustomers+nbVehicles+1..nbCustomers+2*nbVehicles-1)) (
53         successor[n] = n-nbVehicles+1
54     );
55
56 constraint successor[nbCustomers+2*nbVehicles] = nbCustomers+1;
57
58 constraint
59     forall(n in START_DEPOT_NODES) (
60         vehicle[n] = n-nbCustomers
61     );
62
63 constraint
64     forall(n in END_DEPOT_NODES) (
65         vehicle[n] = n-nbCustomers-nbVehicles
66     );
67
68 constraint
69     forall(n in START_DEPOT_NODES) (
70         arrivalTime[n] = 0
71     );
72
73 constraint
74     forall(n in START_DEPOT_NODES) (
75         load[n] = 0
76     );
77
78 constraint redundant_constraint(
79     forall(n in NODES) (

```

```

80     successor[predecessor[n]] = n
81 )
82 );
83
84 constraint redundant_constraint(
85     forall(n in NODES) (
86         predecessor[successor[n]] = n
87     )
88 );
89
90 constraint circuit(successor);
91 constraint redundant_constraint(
92     circuit(predecessor)
93 );
94
95 constraint redundant_constraint(
96     forall(n in CUSTOMER) (
97         vehicle[predecessor[n]] = vehicle[n]
98     )
99 );
100 constraint
101     forall(n in CUSTOMER) (
102         vehicle[successor[n]] = vehicle[n]
103     );
104
105 constraint
106     forall(n in CUSTOMER) (
107         arrivalTime[n] + distance[n,successor[n]] <= arrivalTime[successor[n]]
108     );
109 constraint
110     forall(n in START_DEPOT_NODES) (
111         arrivalTime[n] + distance[n,successor[n]] <= arrivalTime[successor[n]]
112     );
113
114 constraint
115     forall(n in CUSTOMER) (
116         load[n] + demand[n] = load[successor[n]]
117     );
118
119 % Fault added: successor[n] became predecessor[n]
120 constraint
121     forall(n in START_DEPOT_NODES) (
122         load[n] = load[predecessor[n]]
123     );
124
125 constraint
126     objective = sum (depot in END_DEPOT_NODES) (arrivalTime[depot]);
127
128 solve :: seq_search(
129     [int_search([successor[j] | j in NODES],
130         first_fail, indomain_split, complete),
131     int_search(vehicle, first_fail, indomain_split, complete),

```

```

132   int_search([arrivalTime[j] | j in NODES],
133             first_fail, indomain_min, complete),
134   int_search([load[j] | j in NODES],
135             first_fail, indomain_min, complete)
136   ])
137 minimize objective;

```

C.3 RCMSP

A version of this model is presented in Section 4.5.3. The added fault was shown in Listing 6.6.

C.4 Free-Pizza

```

1  int: n;
2  set of int: PIZZA = 1..n;
3  array[PIZZA] of int: price;
4  int: m;
5  set of int: VOUCHER = 1..m;
6  array[VOUCHER] of int: buy;
7  array[VOUCHER] of int: free;
8  set of int: ASSIGN = -m .. m;
9
10 array[PIZZA] of var ASSIGN: how;
11 array[VOUCHER] of var bool: used;
12
13 constraint forall(v in VOUCHER)
14   (used[v] <-> sum(p in PIZZA) (how[p] = -v) >= buy[v]);
15 % Fault added: < became <=
16 constraint forall(v in VOUCHER)
17   (sum(p in PIZZA) (how[p] = -v) < used[v]*buy[v]);
18
19 constraint forall(v in VOUCHER)
20   (sum(p in PIZZA) (how[p] = v) <= used[v]*free[v]);
21
22 constraint forall(p1, p2 in PIZZA)
23   ((how[p1] < how[p2] /\ how[p1] = -how[p2])
24    -> price[p2] <= price[p1]);
25
26 int: total = sum(price);
27 var 0..total: objective = sum(p in PIZZA) ((how[p] <= 0)*price[p]);
28
29 solve :: int_search(how, input_order, indomain_min, complete)
30   minimize objective;

```

C.5 Mapping

```

1  include "globals.mzn";
2
3  int: row;
4  int: col;
5  int: k = row*col;
6  int: n = k + 2;
7  int: no_links;
8  int: m = no_links + 2*k;
9  int: no_flows;
10 int: link_bandwidth;
11 int: no_actors;
12 array[1..m, 1..2] of int: arc;
13 array[1..m] of int: unit_cost;
14 array[1..no_flows, 1..n] of int: balance;
15 int: processor_load;
16
17 array[1..k] of var 0..processor_load: cpu_loads;
18
19 array[1..no_actors] of int: actor_load;
20 array[1..2, 1..no_flows] of int: load;
21
22 array[1..no_flows] of var 0..10000: cost;
23 var 0..10000: communication_cost;
24 var 0..10000: objective;
25 array[1..2*no_flows] of var 1..k: flow_processor;
26 array[1..no_links] of var 0..link_bandwidth : comm_full;
27 array[1..k] of var 0..5*link_bandwidth : flow_from_processor;
28
29 array[1..k] of var 0..10000*no_flows: cc;
30
31 array[1..no_flows] of int: inStream;
32 array[1..no_flows, 1..n] of int: b;
33
34 array[1..no_flows, 1..k] of var 0..max(inStream): inFlow;
35 array[1..no_flows, 1..k] of var 0..max(inStream): outFlow;
36 array[1..no_flows, 1..no_links] of var 0..link_bandwidth: commFlow;
37
38 array[1..no_flows, 1..m] of
39     var 0..max(max(inStream), link_bandwidth): flows;
40
41 array[1..no_flows, 1..2] of int: source_destination_actor;
42
43 array[1..no_actors] of var 1..k: actor_processor;
44
45 array[1..2*k*no_flows + no_links] of
46     var 0..max(max(inStream), link_bandwidth): total_flow;
47 array[1..2*k*no_flows + no_links] of int: total_unit_cost;
48 array[1..k+2*no_flows] of int: total_balance;
49
50 array[1..2*no_flows*k] of int: in_connections;

```

```

51 array[1..2*no_flows*k] of int: out_connections;
52 array[1..2*no_flows*k+no_links, 1..2] of int: all_connections;
53
54 constraint
55   forall( i in 1..no_flows, j in 1..k) (inFlow[i,j] in {0, inStream[i]}
56     /\ outFlow[i,j] in {0, inStream[i]});
57
58 constraint
59   forall (i in 1..no_flows) (count([flows[i,j] | j in 1..k], 0, k-1)
60     /\ count([flows[i,j] | j in m-k+1..m], 0, k-1));
61
62 constraint
63   forall (i in 1..no_flows)
64     (network_flow_cost( arc, [balance[i,j] | j in 1..n],
65       unit_cost, [flows[i,j] | j in 1..m], cost[i]));
66
67 constraint
68   network_flow_cost(all_connections, total_balance, total_unit_cost,
69     total_flow, communication_cost)
70   /\
71   forall (i in 1..no_links)
72     (comm_full[i] = sum(j in 1..no_flows) (flows[j,i+k]));
73
74 % Fault added: flows[i,j] != 0 became flows[i,j] != 1
75 constraint
76   forall (i in 1..no_flows, j in 1..k)
77     ( ( flows[i,j] != 0 <-> flow_processor[i] = j)
78     /\
79     ( flows[i, m-k+j] != 1 <-> flow_processor[no_flows+i] = j) )
80   /\
81   bin_packing_load(cpu_loads, flow_processor,
82     [load[i, j] | i in 1..2, j in 1..no_flows]);
83
84 constraint
85   forall (i in 1..k)
86     (cc[i] = sum(j in 1..no_flows)
87       ((flow_processor[source_destination_actor[j,1]] = i) * cost[j]))
88   /\
89   maximum(objective, [cpu_loads[i] + cc[i] | i in 1..k]);
90
91 constraint
92   forall (i in 1..k)
93     (flow_from_processor[i] =
94       sum(j in 1..no_flows, n in k+1..no_links+k where arc[n,1] = i)
95       (flows[j,n]));
96
97 constraint
98   forall ( i in 1..no_flows) (
99     forall (j in i+1..no_flows
100       where source_destination_actor[i,1] = source_destination_actor[j,1])
101       (flow_processor[i] = flow_processor[j]))
102   /\

```



```

103 forall ( i in 1..no_flows) (
104   forall (j in 1..no_flows
105     where source_destination_actor[i,1] = source_destination_actor[j,2])
106     (flow_processor[i] = flow_processor[no_flows+j]))
107 /\
108 forall ( i in 1..no_flows) (
109   forall (j in i+1..no_flows
110     where source_destination_actor[i,2] = source_destination_actor[j,2])
111     (flow_processor[no_flows+i] = flow_processor[no_flows+j]));
112
113 solve ::
114   seq_search([
115     int_search(flow_processor, first_fail, indomain_min, complete),
116     int_search(total_flow, smallest, indomain_min, complete),
117     int_search(cost ++ [communication_cost], input_order,
118       indomain_min, complete),
119     int_search( cpu_loads, first_fail, indomain_min, complete),
120     int_search( [inFlow[i,j] | i in 1..no_flows, j in 1..k],
121       first_fail, indomain_min, complete),
122     int_search( [outFlow[i,j] | i in 1..no_flows, j in 1..k],
123       first_fail, indomain_min, complete),
124     int_search( [commFlow[i,j] | i in 1..no_flows, j in 1..no_links],
125       first_fail, indomain_min, complete),
126     int_search( [flows[i,j] | i in 1..no_flows, j in 1..m],
127       first_fail, indomain_min, complete),
128     int_search( actor_processor, first_fail, indomain_min, complete)
129   ])
130 minimize objective;

```

C.6 MKnapsack

```

1 include "knapsack.mzn";
2
3 int: N;
4 int: M;
5 set of int: VRange = 1..N;
6 set of int: CRange = 1..M;
7 array[CRange,VRange] of int: a;
8 array[CRange] of int: b;
9 array[VRange] of int: c;
10 int: z;
11
12 array[VRange] of var 0..1: x;
13 array[CRange] of var 0..ub_array(b): bVar;
14 var 0..sum(c): objective = sum(i in VRange) (c[i] * x[i]);
15
16 % Fault added: >= became >
17 constraint
18   forall(i in CRange) ( bVar[i] > 0 /\ bVar[i] <= b[i] );
19
20 constraint

```

```

21 forall(i in CRange) (
22     knapsack(c, [a[i,j] | j in VRange], x, bVar[i], z)
23 );
24
25 solve
26 :: int_search(x, input_order, indomain_max, complete)
27 maximize objective;

```

C.7 NMSeq

```

1 int: n;
2
3 array [1..n] of var 0..n-1: s;
4
5 % Fault added: xs[i] = v became xs[i] != v
6 predicate number_of_v(array [int] of var int: xs, var int: x, int: v) =
7     x == sum (i in index_set(xs)) (bool2int(xs[i] != v));
8
9 constraint
10 forall (i in index_set(s)) (
11     number_of_v(s, s[i], i-1)
12 );
13
14 solve :: int_search(s, input_order, indomain_min, complete)
15 satisfy;

```

C.8 Open-Stacks

```

1 include "globals.mzn";
2
3 int: c;
4 int: p;
5 set of int: Custs = 1..c;
6 set of int: Prods = 1..p;
7 array[Custs,Prods] of 0..1: orders;
8 array[Custs] of int: norders =
9     [ sum(j in Prods) (orders[i,j]) | i in Custs ];
10
11 array[Prods] of var Prods: s;
12 array[Custs,0..p] of var 0..max(norders): o ;
13 var 0..c: objective;
14
15 constraint alldifferent(s);
16
17 % Fault added: o[i,0] became o[i,1]
18 constraint forall (i in Custs) (o[i,1] = 0);
19
20 constraint forall(t in 1..p, i in Custs)
21     (o[i,t] = o[i,t-1] + orders[i,s[t]]);

```

```

22
23 constraint objective =
24     max(j in Prods) (
25         sum(i in Custs) (
26             bool2int( o[i,j-1] < norders[i] /\ o[i,j] > 0 )
27         )
28     );
29
30 solve ::
31     int_search(s, input_order, indomain_min, complete)
32     minimize objective;

```

C.9 P1F

```

1  include "globals.mzn";
2
3  predicate union_circuit( array[int] of var int: x,
4      array[int] of var int: y) =
5
6      let { int: n = length(x),
7          array[1..n] of var 1..n: c
8      } in
9
10     forall(i in 1..n) (
11         let {
12             var 1..2: index
13         } in
14         [x[i], y[i]][index] = c[i]
15     ) /\
16
17     circuit(c) /\
18
19     symmetry_breaking_constraint(c[1] == x[1]);
20
21 int: n;
22 int: m = n-1;
23 array[1..m, 1..n] of var 1..n: p;
24 int: obj_lb = sum(i in 1..n) (i);
25 int: obj_ub = sum(i in 1..n) (i * n);
26
27 var obj_lb..obj_ub: objective = sum(i in 1..n) (i * p[1, i]);
28
29 % Fault added: p[row, col] != col became p[row, col] == col
30 constraint
31     forall(row in 1..m) (
32         forall(col in 1..n) (
33             p[row, col] == col
34         )
35     );
36
37 constraint

```

```

38 forall(row in 1..m) (
39   let {
40
41     array[1..n] of var int: ra = [p[row, i] | i in 1..n]
42   } in
43     inverse(ra, ra) :: domain
44   );
45
46 constraint
47   forall(col in 1..n) (
48     let {
49
50       array[1..m] of var int: c = [p[i, col] | i in 1..m]
51     } in
52       alldifferent(c) :: domain
53     );
54
55 constraint
56   forall(rowa,rowb in 1..m where rowa<rowb) (
57     let {
58       array[1..n] of var int: ra = [p[rowa, i] | i in 1..n],
59       array[1..n] of var int: rb = [p[rowb, i] | i in 1..n]
60     } in
61       union_circuit(ra, rb) :: domain
62     );
63
64 constraint
65   forall(row in 1..m-1) (
66     symmetry_breaking_constraint(
67       let {
68         array[1..n] of var int: ra = [p[row , i] | i in 1..n],
69         array[1..n] of var int: rb = [p[row+1, i] | i in 1..n]
70       } in
71         lex_less(ra, rb)
72     )
73   );
74
75 solve
76   :: int_search(
77     [p[i,j] | i in 1..m, j in 1..n],
78     input_order, indomain_min, complete
79   )
80   minimize objective;

```

C.10 Radiation

```

1  int: m;
2  int: n;
3  set of int: Rows    = 1..m;
4  set of int: Columns = 1..n;
5  array[Rows, Columns] of int: Intensity;
6  set of int: BTimes = 1..Bt_max;
7  int: Bt_max = max(i in Rows, j in Columns) (Intensity[i,j]);
8  int: Ints_sum = sum(i in Rows, j in Columns) (Intensity[i,j]);
9
10 var 0..Ints_sum: Beamtime;
11 var 0..m*n: K;
12 array[BTimes] of var 0..m*n: N;
13 array[Rows, Columns, BTimes] of var 0..m*n: Q;
14
15 constraint
16   Beamtime = sum(b in BTimes) (b * N[b])
17   /\
18   K = sum(b in BTimes) (N[b])
19   /\
20   % Fault added: b * Q[i,j,b] became i * Q[i,j,b]
21   forall(i in Rows, j in Columns)
22     ( Intensity[i,j] = sum([i * Q[i,j,b] | b in BTimes]) )
23   /\
24   forall(i in Rows, b in BTimes)
25     ( upper_bound_on_increments(N[b], [Q[i,j,b] | j in Columns]) );
26
27 predicate upper_bound_on_increments(var int: N_b,
28                                     array[int] of var int: L) =
29   N_b >= L[1] + sum([ max(L[j] - L[j-1], 0) | j in 2..n ]);
30
31 int: obj_min = lb((m*n + 1) * Beamtime + K);
32 int: obj_max = ub((m*n + 1) * Beamtime + K);
33 var obj_min..obj_max: objective = (m*n + 1) * Beamtime + K;
34
35 solve
36   :: int_search(
37     [Beamtime] ++ N ++
38     [Q[i,j,b] | i in Rows, j in Columns, b in BTimes ],
39     input_order, indomain_split, complete)
40   minimize objective;

```

C.11 Spot5

```

1  include "table.mzn";
2
3  int: num_variables;
4  int: min_domain;
5  int: max_domain;
6  array[1..num_variables] of set of int: domains;
7  array[1..num_variables] of int: costs;
8  int: num_constraints2;
9  int: max_constraints2;
10 int: num_constraints3;
11 int: max_constraints3;
12 array[1..num_constraints2] of int: scopes2x;
13 array[1..num_constraints2] of int: scopes2y;
14 array[1..max_constraints2] of int: constraints2;
15 array[1..num_constraints2] of int: num_tuples2;
16 array[1..num_constraints2] of int: cum_tuples2;
17 array[1..num_constraints3] of int: scopes3x;
18 array[1..num_constraints3] of int: scopes3y;
19 array[1..num_constraints3] of int: scopes3z;
20 array[1..max_constraints3] of int: constraints3;
21 array[1..num_constraints3] of int: num_tuples3;
22 array[1..num_constraints3] of int: cum_tuples3;
23
24 array[1..num_variables] of var min_domain..max_domain: p;
25 int: obj_min = sum(j in 1..num_variables)
26     (if costs[j] < 0 then costs[j] else 0 endif);
27 int: obj_max = sum(j in 1..num_variables)
28     (if costs[j] > 0 then costs[j] else 0 endif);
29 var obj_min..obj_max: objective;
30
31 solve :: seq_search([
32     int_search(p, first_fail, indomain_min, complete),
33     int_search([objective], input_order, indomain_min, complete)
34 ]) minimize objective;
35
36 constraint objective = sum(j in 1..num_variables)
37     ( costs[j] * bool2int(p[j] = 0) );
38
39 constraint forall(j in 1..num_variables) (
40     p[j] in domains[j]);
41
42 % Fault added: constraints2[u] became constraints2[u]+1
43 constraint
44     forall(j in 1..num_constraints2) (
45         table(
46             [p[scopes2x[j]], p[scopes2y[j]]],
47             array2d(1..num_tuples2[j], 1..2,
48                 [ constraints2[u]+1
49                   | u in (2*cum_tuples2[j]+1)..(2*cum_tuples2[j]+num_tuples2[j]*2)]
50             )

```

```

51     )
52 );
53
54 constraint
55     forall(j in 1..num_constraints3) (
56         table(
57             [p[scopes3x[j]], p[scopes3y[j]], p[scopes3z[j]]],
58             array2d(1..num_tuples3[j],1..3,
59                 [ constraints3[u]
60                   | u in (3*cum_tuples3[j]+1)..(3*cum_tuples3[j]+num_tuples3[j]*3)]
61             )
62         )
63 );

```

C.12 TDTSP

```

1  include "inverse.mzn";
2
3  int: n;
4  set of int: VISIT = 1..n;
5  set of int: VISIT1 = 1..n+1;
6  array[VISIT] of int: D;
7  int: q;
8  set of int: PREC = 1..q;
9  array[PREC,1..2] of VISIT: prec;
10 int: steps;
11 int: l;
12 int: H = steps * l - 1;
13 set of int: TIME = 0..H;
14 int: r;
15 set of int: FORBID = 1..r;
16 array[FORBID] of VISIT: which;
17 array[FORBID,1..2] of TIME: interval;
18 array[VISIT1,VISIT1,0..steps] of int: T;
19
20 array[VISIT1] of var VISIT1: position;
21 array[VISIT1] of var VISIT1: next;
22 array[VISIT1] of var VISIT1: prev;
23 array[VISIT1] of var TIME: tvar;
24 var TIME: objective = tvar[n+1];
25 array[VISIT1] of var VISIT1: atposition;
26
27 constraint inverse(next,prev);
28 constraint position[1] = 1;
29
30 % Fault added: n+1 became n
31 constraint position[n+1] = n;
32 constraint tvar[1] = 0;
33 constraint prev[1] = n+1 /\ next[n+1] = 1;
34
35 constraint forall(i in VISIT1) (next[i] != i /\ prev[i] != i);

```

```

36
37 constraint forall(i in VISIT) (position[next[i]] = position[i] + 1);
38 constraint forall(i in 2..n+1) (position[i] = position[prev[i]] + 1);
39
40 constraint forall(i in 2..n+1) (
41     tvar[i] >= tvar[prev[i]] + D[prev[i]] +
42     T[prev[i],i,tvar[prev[i]] div 1]
43 );
44
45 constraint forall(i in 1..n) (
46     tvar[next[i]] >= tvar[i] + D[i] + T[i,next[i],tvar[i] div 1]
47 );
48
49 constraint forall(i in PREC) (
50     tvar[prec[i,1]] + D[prec[i,1]] <= tvar[prec[i,2]]
51 );
52
53 constraint forall(i in FORBID) (
54     tvar[which[i]] + D[which[i]] <= interval[i,1]
55     /\  tvar[which[i]] >= interval[i,2]
56 );
57
58 constraint redundant_constraint (
59     tvar[n+1] >= sum(D) + sum(i in VISIT)
60     (T[i,next[i],tvar[i] div 1])
61 );
62 constraint redundant_constraint (
63     tvar[n+1] >= sum(D) + sum(i in 2..n+1)
64     (T[prev[i],i,tvar[prev[i]] div 1])
65 );
66
67 constraint redundant_constraint (inverse(position,atposition));
68 constraint redundant_constraint (atposition[1] = 1);
69 constraint redundant_constraint (atposition[n+1] = n+1);
70 constraint redundant_constraint (
71     forall(j in VISIT) (next[atposition[j]] = atposition[j+1])
72 );
73 constraint redundant_constraint (
74     forall(j in VISIT) (prev[atposition[j+1]] = atposition[j])
75 );
76
77 solve
78     :: seq_search([
79         int_search(atposition, input_order, indomain_min, complete),
80         int_search(tvar, smallest, indomain_min, complete)
81     ])
82 minimize objective;

```


Vita

Publications arising from this thesis include:

Kevin Leo and Guido Tack

“Debugging Unsatisfiable Constraint Models”. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Proceedings*, pp. 77–93.

Kevin Leo and Guido Tack

“Multi-Pass High-Level Presolving”. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pp. 346–352.

Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda

“Globalizing constraint models”. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013. Proceedings*, pp. 432–447.

Permanent Address: Caulfield School of Information Technology
Monash University
Australia

This thesis was typeset with $\text{\LaTeX} 2_{\epsilon}$ ¹ by the author.

¹ $\text{\LaTeX} 2_{\epsilon}$ is an extension of \LaTeX . \LaTeX is a collection of macros for \TeX . \TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

Bibliography

- Abío, I., Stuckey, P. J. (2014). “Encoding Linear Constraints into SAT”. *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Proceedings*, pp. 75–91.
- Achterberg, T. (2009). “SCIP: Solving constraint integer programs”. *Mathematical Programming Computation* 1, pp. 1–41.
- Andersen, E. D., Andersen, K. D. (1995). “Presolving in linear programming”. *Mathematical Programming* 2, pp. 221–245.
- Ansótegui, C., Manyà, F. (2005). “Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables”. *The Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, Proceedings*, pp. 1–15.
- Audemard, G., Hoessen, B., Jabbour, S., Piette, C. (2014). “Dolius: A Distributed Parallel SAT Solving Framework”. *POS-14. Fifth Pragmatics of SAT workshop*, pp. 1–11.
- Bacchus, F., Katsirelos, G. (2016). “Finding a Collection of MUSes Incrementally”. *Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Proceedings*, pp. 35–44.
- Bailey, J., Stuckey, P. J. (2005). “Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization”. *Practical Aspects of Declarative Languages: 7th International Symposium, PADL 2005, Proceedings*, pp. 174–186.
- Barahona, F., Anbil, R. (2000). “The volume algorithm: producing primal solutions with a subgradient method”. *Mathematical Programming* 3, pp. 385–399.
- Bardin, S., Gotlieb, A. (2012). “FDCC: A Combined Approach for Solving Constraints over Finite Domains and Arrays”. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 9th International Conference, CPAIOR 2012, Proceedings*, pp. 17–33.
- Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T. (2007). “Global Constraint Catalogue: Past, Present and Future”. *Constraints* 1, pp. 21–62.

- Beldiceanu, N., Simonis, H. (2011). “A Constraint Seeker: Finding and Ranking Global Constraints from Examples”. *Principles and Practice of Constraint Programming, 17th International Conference, CP 2011, Proceedings*, pp. 12–26.
- (2012). “A Model Seeker: Extracting Global Constraint Models from Positive Examples”. *Principles and Practice of Constraint Programming, 18th International Conference, CP 2012, Proceedings*, pp. 141–157.
- Belov, G., Stuckey, P. J., Tack, G., Wallace, M. (2016). “Improved Linearization of Constraint Programming Models”. *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Proceedings*, pp. 49–65.
- Bergen, M. E., Beek, P. van, Carchrae, T. (2001). “Constraint-Based Vehicle Assembly Line Sequencing”. *Advances in Artificial Intelligence: 14th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI 2001, Proceedings*, pp. 88–99.
- Bessiere, C., Coletta, R., Koriche, F., O’Sullivan, B. (2005a). “A SAT-based version space algorithm for acquiring constraint satisfaction problems”. *Machine Learning: ECML 2005*, pp. 23–34.
- Bessiere, C., Coletta, R., Petit, T. (2005b). “Acquiring parameters of implied global constraints”. *Principles and Practice of Constraint Programming, 11th International Conference, CP 2005, Proceedings*, pp. 747–751.
- (2007). “Learning implied global constraints”. *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pp. 44–49.
- Bessiere, C., Cardon, S., Debruyne, R., Lecoutre, C. (2011). “Efficient algorithms for singleton arc consistency”. *Constraints* 1, pp. 25–53.
- Biere, A., Le Berre, D., Lonca, E., Manthey, N. (2014). “Detecting Cardinality Constraints in CNF”. *Theory and Applications of Satisfiability Testing, 17th International Conference, SAT 2014, Proceedings*. Ed. by C. Sinz, U. Egly, pp. 285–301.
- Bockmayr, A., Pissaruk, N. (2006). “Detecting Infeasibility and Generating Cuts for Mixed Integer Programming Using Constraint Programming”. *Computers and Operations Research* 10, pp. 2777–2786.
- Brand, S., Duck, G. J., Puchinger, J., Stuckey, P. J. (2008). “Flexible, Rule-Based Constraint Model Linearisation”. *Practical Aspects of Declarative Languages*, pp. 68–83.

- Charnley, J., Colton, S., Miguel, I. (2006a). “Automatic Generation of Implied Constraints”. *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pp. 73–77.
- (2006b). “Automatic generation of implied constraints”. *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pp. 73–77.
- Cheng, B. M. W., Lee, J. H., Wu, J. C. K. (1996). “Speeding Up Constraint Propagation By Redundant Modeling”. *Principles and Practice of Constraint Programming, Second International Conference, CP 1996, Proceedings*, pp. 91–103.
- Cheng, B. M. W., Choi, K. M. F., Lee, J. H., Wu, J. C. K. (1999). “Increasing Constraint Propagation by Redundant Modeling: an Experience Report”. *Constraints* 4, pp. 167–192.
- Choi, C. W., Lee, J. H., Stuckey, P. J. (2004). “Removing Propagation Redundant Constraints in Redundant Modeling”. *CoRR*.
- Chu, G. (2011). “Improving Combinatorial Optimization”. PhD thesis. The University of Melbourne.
- Chu, G., Stuckey, P. J. (2012a). “A Generic Method for Identifying and Exploiting Dominance Relations”. *Principles and Practice of Constraint Programming, 18th International Conference, CP 2012, Proceedings*, pp. 6–22.
- (2012b). “Inter-instance Nogood Learning in Constraint Programming”. *Principles and Practice of Constraint Programming, 18th International Conference, CP 2012, Proceedings*, pp. 238–247.
- Coffman Jr., E. G., Garey, M. R., Johnson, D. S. (1997). “Approximation Algorithms for NP-hard Problems”. Chap. Approximation Algorithms for Bin Packing: A Survey, pp. 46–93.
- COIN-OR (2016). *COIN-OR CBC - Coin-Or Branch and Cut*. <https://projects.coin-or.org/Cbc>.
- Cook, S. A. (1971). “The Complexity of Theorem-proving Procedures”. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC 1971, pp. 151–158.
- Crowder, H., Johnson, E. L., Padberg, M. (1983). “Solving Large-Scale Zero-One Linear Programming Problems”. *Operations Research* 31, pp. 803–834.
- Dakin, R. J. (1965). “A tree-search algorithm for mixed integer programming problems”. *The Computer Journal* 3, pp. 250–255.

- Dantzig, G. (1963). *Linear Programming and Extensions*. Princeton landmarks in mathematics and physics.
- Davis, M., Logemann, G., Loveland, D. (1962). “A Machine Program for Theorem-proving”. *Commun. ACM* 7, pp. 394–397.
- Dekker, J. J. (2016). “Sub-Problem Pre-Solving in MiniZinc”. MA thesis. Uppsala University, Department of Information Technology.
- Dincbas, M., Simonis, H., Hentenryck, P. V. (1988). “Solving the Car Sequencing Problem in Constraint Logic Programming”. In *European Conference on Artificial Intelligence, ECAI 1988*, pp. 290–295.
- Dittel, A., Fuegenschuh, A., Goettlich, S., Herty, M. (2009). “MIP presolve techniques for a PDE-based supply chain model”. *Optimization Methods & Software* 24, pp. 427–445.
- Dongen, M. R. C. van (2006). “Beyond Singleton Arc Consistency”. *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pp. 163–167.
- Eén, N., Sörensson, N. (2004). “An Extensible SAT-solver”. *Theory and Applications of Satisfiability Testing*. Chap. 37, pp. 333–336.
- Eén, N., Biere, A. (2005). “Effective Preprocessing in SAT Through Variable and Clause Elimination”. *Theory and Applications of Satisfiability Testing*, pp. 61–75.
- Emden, M. H. van, Kowalski, R. A. (1976). “The Semantics of Predicate Logic As a Programming Language”. *J. ACM* 4, pp. 733–742.
- Fang, R. J. F., Sandrin, W. A. (1977). “Carrier frequency assignment for nonlinear repeaters”. *COMSAT Technical Review*, pp. 227–245.
- Feydy, T., Stuckey, P. J. (2009). “Lazy Clause Generation Reengineered”. *Principles and Practice of Constraint Programming, 15th International Conference, CP 2009, Proceedings*, pp. 352–366.
- Feydy, T., Somogyi, Z., Stuckey, P. J. (2011). “Half Reification and Flattening”. *Principles and Practice of Constraint Programming, 17th International Conference, CP 2011, Proceedings*, pp. 286–301.
- FICO (2016). *FICO Express*. <http://www.fico.com/en/products/fico-xpress-optimization-suite>.
- Fischetti, M., Monaci, M. (2014). “Exploiting Erraticism in Search”. *Operations Research* 62, pp. 114–122.

- Fourer, R., Gay, D. M., Kernighan, B. W. (1989). *AMPL: A Mathematical Programming Language*. Tech. rep. Management Science.
- Freuder, E. C. (1997). “In Pursuit of the Holy Grail”. *Constraints* 1, pp. 57–61.
- Frisch, A. M., Miguel, I., Walsh, T. (2001). “Extensions to Proof Planning for Generating Implied Constraints”. *Calculemus-01*, pp. 130–141.
- (2003). “CGRASS: A system for transforming constraint satisfaction problems”. *Recent Advances in Constraints*, pp. 15–30.
- Frisch, A. M., Jefferson, C., Martínez, B. H., Miguel, I. (2005). “The rules of constraint modelling”. *International Joint Conference on Artificial Intelligence*, pp. 109–116.
- Frisch, A. M., Grum, M., Jefferson, C., Martínez, B. H., Miguel, I. (2007). “The design of ESSENCE: a constraint language for specifying combinatorial problems”. *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pp. 80–87.
- Frisch, A. M., Stuckey, P. J. (2009). “The Proper Treatment of Undefinedness in Constraint Languages”. *Principles and Practice of Constraint Programming, 15th International Conference, CP 2009, Proceedings*, pp. 367–382.
- Gamrath, G., Fischer, T., Gally, T., Gleixner, A. M., Hendel, G., Koch, T., Maher, S. J., Miltenberger, M., Müller, B., Pfetsch, M. E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Vigerske, S., Weninger, D., Winkler, M., Witt, J. T., Witzig, J. (2016). *The SCIP Optimization Suite 3.2*. Tech. rep. ZIB.
- Garcia de la Banda, M., Marriott, K., Rafeh, R., Wallace, M. (2006). “The Modelling Language Zinc”. *Principles and Practice of Constraint Programming, 12th International Conference, CP 2006, Proceedings*, pp. 700–705.
- Gasca, R. M., Valle, C., Gómez-López, M. T., Ceballos, R. (2007). “NMUS: Structural Analysis for Improving the Derivation of All MUSEs in Overconstrained Numeric CSPs”. *Current Topics in Artificial Intelligence: 12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007*, pp. 160–169.
- Gecode Team (2006). *Gecode: Generic Constraint Development Environment*. <http://www.gecode.org>.
- Gent, I. P. (2002). “Arc Consistency in SAT”. *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002*, pp. 121–125.
- Gent, I. P., Walsh, T. (1999). *CSPLib: a benchmark library for constraints*. Tech. rep. <http://www.csplib.org>.

- Gent, I. P., Jefferson, C., Miguel, I. (2006a). “MINION: A Fast, Scalable, Constraint Solver”. *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pp. 98–102.
- Gent, I. P., Petrie, K. E., Puget, J. (2006b). “Symmetry in Constraint Programming”. *Handbook of Constraint Programming*, pp. 329–376. ISBN: 978-0-444-52726-4.
- Gent, I. P., Miguel, I., Rendl, A. (2007a). “Tailoring solver-independent constraint models: A case study with Essence’ and Minion”. *Abstraction, Reformulation, and Approximation*, pp. 184–199.
- (2007b). “Tailoring solver-independent constraint models: a case study with ESSENCE’ and MINION”. *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation. SARA 2007*, pp. 184–199.
- Gent, I. P., Miguel, I., Nightingale, P. (2008). “Generalised arc consistency for the AllDifferent constraint: An empirical survey”. *Artificial Intelligence* 18, pp. 1973–2000.
- Gleeson, J., Ryan, J. (1990). “Identifying Minimally Infeasible Subsystems of Inequalities”. *INFORMS Journal on Computing* 1, pp. 61–63.
- Gondzio, J. (1997). “Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method”. *INFORMS Journal on Computing* 9, pp. 73–91.
- Gurobi Optimization, Inc. (2015). *Gurobi Optimizer Reference Manual*. <http://www.gurobi.com>.
- Güsgen, H., Hertzberg, J. (1992). *A perspective of constraint-based reasoning: an introductory tutorial*.
- Hebrard, E. (2016). *Mistral 2.0*. <https://github.com/ehebrard/Mistral-2.0>.
- Heinz, S., Schulz, J., Beck, J. C. (2013). “Using dual presolving reductions to reformulate cumulative constraints”. *Constraints* 18, pp. 166–201.
- Hoffman, K. L., Padberg, M. (1991). “Improving LP-Representations of Zero-One Linear Programs for Branch-and-Cut”. *INFORMS Journal on Computing* 2, pp. 121–134.
- IBM (2010). *IBM ILOG CPLEX Optimizer*. <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>.
- IBM (2016). *IBM ILOG CPLEX CP Optimizer*. <http://www.ibm.com/software/commerce/optimization/cplex-cp-optimizer/>.

- Junker, U. (2001). “QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms”. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*.
- Jussien, N., Ouis, S. (2001). “User-friendly explanations for constraint programming”. *Proceedings of the Eleventh Workshop on Logic Programming Environments, WLPE 2001*.
- Land, A. H., Doig, A. G (1960). “An Automatic Method of Solving Discrete Programming Problems”. *Econometrica* 3, pp. 497–520.
- Law, Y. C., Lee, J. H. (2002). “Model Induction: A New Source of CSP Model Redundancy”. *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pp. 54–61.
- Lazaar, N., Gotlieb, A., Lebbah, Y. (2012). “A CP framework for testing CP”. *Constraints* 17, pp. 123–147.
- Leconte, M. (1996). “A Bounds-Based Reduction Scheme for Difference Constraints”. *Constraints* 96.
- Leo, K., Mears, C., Tack, G., Garcia de la Banda, M. (2013). “Globalizing Constraint Models”. *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Proceedings*, pp. 432–447.
- Leo, K., Tack, G. (2015). “Multi-Pass High-Level Presolving”. *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pp. 346–352.
- (2017). “Debugging Unsatisfiable Constraint Models”. *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Proceedings*, pp. 77–93.
- Levin, L. A. (1973). “Universal search problems”. *Problems of Information Transmission* 3.
- Li, C. M., Anbulagan (1997). “Look-Ahead Versus Look-Back for Satisfiability Problems”. *Principles and Practice of Constraint Programming, Third International Conference, CP 1997, Proceedings*, pp. 341–355.
- Liberto, G. D., Kadioglu, S., Leo, K., Malitsky, Y. (2016). “DASH: Dynamic Approach for Switching Heuristics”. *European Journal of Operational Research* 248, pp. 943–953.

- Liffiton, M. H., Malik, A. (2013). “Enumerating Infeasibility: Finding Multiple MUSes Quickly”. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Proceedings*, pp. 160–175.
- Liffiton, M. H., Previti, A., Malik, A., Marques-Silva, J. (2015). “Fast, flexible MUS enumeration”. *Constraints* 2, pp. 223–250.
- Linderroth, J. T., Savelsbergh, M. W. P. (1999). “A Computational Study of Search Strategies for Mixed Integer Programming”. *INFORMS Journal on Computing* 2, pp. 173–187.
- Loon, J. van (1981). “Irreducibly inconsistent systems of linear inequalities”. *European Journal of Operational Research* 3, pp. 283–288.
- López-Ortiz, A., Quimper, C., Tromp, J., Beek, P. van (2003). “A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint”. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003*, pp. 245–250.
- Mahajan, A. (2010). “Presolving Mixed-Integer Linear Programs”. *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc.
- Makhorin, A (2001). “GLPK-The GNU linear programming toolkit”. <http://www.gnu.org/software/glpk/glpk.html>.
- Mears, C., Garcia de la Banda, M., Wallace, M. (2006). “On implementing symmetry detection”. *Proceedings of the The Sixth International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon 2006*.
- Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B. (2008). “A Novel Approach For Detecting Symmetries in CSP Models”. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, Proceedings*, pp. 158–172.
- Mears, C., Garcia de la Banda, M., Wallace, M. (2009). “On implementing symmetry detection”. *Constraints* 14.4, pp. 443–477.
- Metodi, A., Codish, M., Lagoon, V., Stuckey, P. J. (2011). “Boolean Equi-propagation for Optimized SAT Encoding”. *CoRR* abs/1104.4617.
- MOSEK ApS (2016). *The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 60)*. <http://docs.mosek.com/7.1/toolbox/index.html>.

- Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S. (2001). “Chaff: Engineering an Efficient SAT Solver”. *Proceedings of the 38th Annual Design Automation Conference*. DAC 2001, pp. 530–535.
- Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., Tack, G. (2007). “MiniZinc: Towards a Standard CP Modelling Language”. *Principles and Practice of Constraint Programming, 13th International Conference, CP 2007, Proceedings*, pp. 529–543.
- Nightingale, P. (2016). personal communication.
- Nightingale, P., Akgün, Ö., Gent, I. P., Jefferson, C., Miguel, I. (2014). “Automatically Improving Constraint Models in Savile Row through Associative-Commutative Common Subexpression Elimination”. *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Proceedings*, pp. 590–605.
- O’Callaghan, B., O’Sullivan, B., Freuder, E. C. (2005). “Generating Corrective Explanations for Interactive Constraint Satisfaction”. *Principles and Practice of Constraint Programming, 11th International Conference, CP 2005, Proceedings*, pp. 445–459.
- Ohrimenko, O., Stuckey, P. J., Codish, M. (2007). “Propagation = Lazy Clause Generation”. *Principles and Practice of Constraint Programming, 13th International Conference, CP 2007, Proceedings*, pp. 544–558.
- Opturion (2016). *Opturion CPX*. <http://www.opturion.com/cpx>.
- Ouis, S., Jussien, N., Boizumault, P. (2003). “k-relevant Explanations for Constraint Programming”. *Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society, FLAIRS 2003*, pp. 192–196.
- Papadimitriou, C. H., Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*.
- Prud’homme, C., Fages, J.-G., Lorca, X. (2016). *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. <http://www.choco-solver.org>.
- Puget, J. (1998). “A Fast Algorithm for the Bound Consistency of alldiff Constraints”. *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pp. 359–366.
- (2005). “Automatic Detection of Variable and Value Symmetries”. *Principles and Practice of Constraint Programming, 11th International Conference, CP 2005, Proceedings*, pp. 475–489.

- Régin, J.-C. (1994). “A Filtering Algorithm for Constraints of Difference in CSP”. *Proceedings of the AAAI National Conference*, pp. 362–367.
- (2013). “Simple Solutions for Complex Problems”. Presentation - Principles and Practice of Constraint Programming - 19th International Conference, CP 2013. Presented upon receiving the ACP Research Excellence Award. <http://cp2013.a4cp.org/slides/a1.pdf>.
- Rendl, A., Miguel, I., Gent, I. P., Gregory, P. (2009). “Common Subexpressions in Constraint Models of Planning Problems”. *SARA 2009*.
- Robinson, J., Bernstein, A. (1967). “A class of binary recurrent codes with limited error propagation”. *IEEE Transactions on Information Theory* 1, pp. 106–113.
- Rossi, F., Beek, P. van, Walsh, T., eds. (2006). *Handbook of Constraint Programming*.
- Salvagnin, D. (2014). “Detecting and Exploiting Permutation Structures in MIPs”. *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Proceedings*, pp. 29–44.
- (2016). “Detecting Semantic Groups in MIP Models”. *Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Proceedings*, pp. 329–341.
- Savelsbergh, M. W. P. (1994). “Preprocessing and Probing Techniques for Mixed Integer Programming Problems”. *INFORMS Journal on Computing* 6, pp. 445–454.
- Selman, B., Kautz, H., Cohen, B. (1995). “Local Search Strategies for Satisfiability Testing”. *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pp. 521–532.
- Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M. (2016). “Learning from Learning Solvers”. *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Proceedings*, pp. 455–472.
- Silva, J. a.P. M., Sakallah, K. A. (1996). “GRASP - A New Search Algorithm for Satisfiability”. *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. ICCAD 1996, pp. 220–227.
- Smith, B. M., Brailsford, S. C., Hubbard, P. M., Williams, H. P. (1996). “The progressive party problem: Integer linear programming and constraint programming compared”. *Constraints* 1, pp. 119–138.
- Stuckey, P. J., Becket, R., Fischer, J. (2010). “Philosophy of the MiniZinc challenge”. *Constraints* 15, pp. 307–316.

- Stuckey, P. J., Tack, G. (2013). “MiniZinc with Functions”. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Proceedings*, pp. 268–283.
- Suhl, U. H., Szymanski, R. (1994). “Supernode processing of mixed-integer models”. *Computational Optimization and Applications* 3, pp. 317–331.
- Sutherland, I. E. (1964). “Sketch Pad a Man-machine Graphical Communication System”. *Proceedings of the SHARE Design Automation Workshop*. DAC 1964, pp. 6.329–6.346.
- Tack, G. (2011). “libmzn – A modular CP infrastructure based on MiniZinc”. *MZN’11, first international MiniZinc workshop*.
- Thompson, A., Moran, J., Swenson, G. (2001). *Interferometry and Synthesis in Radio Astronomy*.
- Van Hentenryck, P., Michel, L., Perron, L., Régin, J.-C. (1999). “Constraint Programming in OPL”. *Principles and Practice of Declarative Programming, International Conference PPDP 1999, Proceedings*, pp. 98–116.
- Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M. (2005). “Compositional Derivation of Symmetries for Constraint Satisfaction”. *Abstraction, Reformulation and Approximation*, pp. 234–247.
- Walsh, T. (2000). “SAT v CSP”. *Principles and Practice of Constraint Programming, 6th International Conference, CP 2000, Proceedings*, pp. 441–456.
- Williams, H. P., Yan, H. (2001). “Representations of the All_Different Predicate of Constraint Satisfaction in Integer Programming”. *INFORMS Journal on Computing* 2, pp. 96–103.
- Wolpert, D. H., Macready, W. G. (1997). “No Free Lunch Theorems for Optimization”. *Transactions on Evolutionary Computation* 1, pp. 67–82.
- Wright, M. H. (2005). “The interior-point revolution in optimization: history, recent developments, and lasting consequences”. *Bulletin of the American Mathematical Society*, pp. 39–56.
- Yunes, T. H., Aron, I. D., Hooker, J. N. (2010). “An Integrated Solver for Optimization Problems”. *Operations Research* 58, pp. 342–356.