Towards Machine-Assisted Reformulation for MiniZinc

PHUC H. VO

UPPSALA UNIVERSITY
Department of Information Technology
Towards Machine-Assisted Reformulation for MiniZinc

Phuc H. Vo
| Võ Hữu-Phúc |
huu-phuc.vo@it.uu.se

November 2019

Division of Computing Science
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Phuc H. Vo
| Võ Hữu-Phúc | 2019
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

Model reformulation plays an important role in improving models, reducing search space so that solutions can be found faster. Hence we categorise model reformulation into three types: a model is reformulated to another model with the same modelling language, a model with non constraint programming standard types is compiled to another model with constraint programming (CP) standard types, and a model is converted to Boolean satisfiability problem (SAT), Mixed-integer programming (MIP), Satisfiability modulo theories (SMT), and Mixed integer linear programming (MILP). Based on these categories, reformulation and compilation could be used in different ways to improve a model such as automatic reformulations, semi-automatic reformulations, MIP to constraint programming, implied constraints, set Constraint Satisfaction Problems (CSPs) to CSPs, string variables to CSP without string variables, symmetry breaking, pre-computation, non-binary to binary translations, and reformulations into SAT, MILP, and SMT. CP is a pervasive and highly successful technology for solving a wide variety of constraint satisfaction problems such as air traffic management, resource allocation, transportation, scheduling, and so on. Model reformulation can have a significant impact on solving time. Techniques from formal methods will be used to provide machine assistance for MiniZinc, which is the high-level modelling language to model CSPs. In this thesis, we present an overview of reformulation focusing on the contributions made in the area of reformulation of CSPs where significant performance improvements have been achieved. Our contributions are as follows: propose criteria and types that categorise model reformulations; we systematically unify and organise the vast literature; contrast and compare different categories of the reformulation for solving CSPs; propose a compiler for counter automata reformulation; ultimately, we propose a plan for future work, we identify the challenges, implement frameworks, and evaluate our experimental results of model reformulations.
Acknowledgments

First of all, I would like to thank all supervisors Dave Clarke, Tobias Wrigstad, Joachim Parrow, Pierre Flener, Justin Pearson, Di Yuan.

During my Ph.D. in Uppsala University, it is my pleasure to having lunches and fikas with many nice and supportive colleagues. Thanks Diep Bui-Phi, Phong Ngo-Tuan, Long, Quy Trinh-Cong, Magnus Norgren, Magnus Lång, Stephan, Elias, Albert, Kim-Anh, Andreas, and David. I am glad to have had you as colleagues.

I would like to thank my papa and mama, firstly, for making me exist, secondly, for raising me to believe in myself, never give up, and lastly, for your love and trust. Con cảm ơn Tía và Mẹ đã sinh ra con, nuôi nấng dạy dỗ con biết tin vào bản thân và đừng bao giờ bỏ cuộc, và luôn ủng hộ con trên đường đời.

Ultimately, special thanks to my wife Tram-Anh and Nam-Phuong for supporting and being my motivation during my Ph.D. education. It would be best in my life when you are with me at all times. Dành tặng cho Nam-Phương, Trâm-Anh, tía, và mẹ. Dedicated to Nam-Phương, Trâm-Anh, and my parents.
Selected papers not included in the thesis


## Contents

1 Introduction ................................................................. 3  
   1.1 Background ............................................................ 3  
   1.2 Research methodology .............................................. 4  
      1.2.1 Motivation ..................................................... 4  
      1.2.2 Objectives .................................................... 5  
      1.2.3 Methodology .................................................. 7  
   1.3 Contributions ....................................................... 7  

2 Reformulations ............................................................ 9  
   2.1 Constraint Programming ............................................ 9  
   2.2 Constraint Satisfaction Problems ................................ 10  
   2.3 Constraint Modelling .............................................. 12  
   2.4 Modelling Languages .............................................. 13  
   2.5 Reformulations of CSPs ............................................ 17  
      2.5.1 General Reformulations ..................................... 19  
      2.5.2 Reformulations by Levels of Automation ................. 21  
      2.5.3 Reformulation by Changing Viewpoint ..................... 22  
      2.5.4 Reformulate Implied Constraints .......................... 24  
      2.5.5 Reformulation by Symmetry Breaking ....................... 25  
      2.5.6 Reformulation by Precomputation .......................... 26  
      2.5.7 Reformulate SET-CSPs to CSPs ............................. 27  
      2.5.8 Reformulate String Variables ............................... 28  
      2.5.9 Reformulate Non-binary to binary translations .......... 28  
      2.5.10 Reformulations into SAT .................................. 29  
      2.5.11 Reformulations into MILP and SMT ....................... 30  

3 Summary of Contributions .............................................. 31  
   3.1 A Survey of Reformulations of Constraint Satisfaction Problems 32  
   3.2 Towards Efficient Solvers for Optimisation Problems .......... 32  
   3.3 Automata with Counters in MiniZinc ............................. 33  
      3.3.1 Describing an Automaton in MiniZinc ....................... 34
3.3.2 Describing an Automata with Counters ............... 35
3.3.3 Reformulation Process .................................. 37
3.4 Machine-Assisted Reformulation for MiniZinc ............. 41

4 Conclusions and Future Work ................................ 45
Chapter 1

Introduction

1.1 Background

Constraint programming (CP) is pervasive and highly successful technology for solving a wide variety of constraint satisfaction problems (CSPs) such as air traffic management, resource allocation, transportation, scheduling, robot task sequencing, sensor network configuration, compiler design, base station testing, school tabling, sports tournament design, container packing, bioinformatics, and so on [7,73,128,139]. A CSP is defined as a decision problem where the goal is to determine whether an assignment of a finite domain of possible values to a finite set of variables exists, which satisfies a given finite set of constraints. Popular constraint solvers include ILOG Solver [124], Choco [121], GECODE solver [69], HAMPI [90,91], KALUZA [131], SUSHI [68], Z3 [45]. There are logic programming based constraint modelling languages and toolchain such as ECLiPSe [10], SICStus Prolog [29], ESSENCE [64,65], and CONJURE [5]. Model reformulation plays an important role in improving models, reducing search space so that solutions can be found faster [112]. Hence we categorise model reformulation into three types: that a model is reformulated to another model with the same modelling language [139,146], that a model with non CP standard types is compiled to another model with CP standard types [74], and that a model is converted to Boolean satisfiability problem (SAT), Mixed-integer programming (MIP), Satisfiability modulo theories (SMT), and Mixed integer linear programming (MILP) [74,109,114]. Based on these categories, reformulation and compilation could be used in different ways to improve a model such as automatic reformulations, semi-automatic reformulations, multiple viewpoints, MIP to CP, implied constraints, set CSPs to CSPs, string variables to CSP without string variables, symmetry breaking, pre-computation, non-binary to binary translations, and reformulations into
SAT, MILP, SMT. In this thesis, we present an overview of reformulation focusing on the contributions made in the area of reformulation of CSPs where significant performance improvements have been achieved. According to criteria and types that categorise reformulation, we systematically unify and organise the vast literature as an in-depth study on the model reformulation of CSPs in CP. In addition, a number of case studies that are empirically studied are sorted into those reformulation and compilation categories. This thesis contrasts and compares different categories of reformulation for solving CSPs as well as ways of applying these categories.

1.2 Research methodology

Model reformulation plays an important role in improving models and reducing search space so that solutions can be found faster. In solving CSPs, a model of a CSP may be solved rapidly, while a different model may take excessively long to solve. The efficient solution of CSP is significant in real-world applications, such as air traffic management, resource allocation, production scheduling, and bioinformatics. Many technologies such as CP, hybrid technologies, mixed integer programming (MIP), constraint-based local search (CBLS), boolean satisfiability (SAT) could have different solvers and backends to solve the real-time problems. Model reformulation can have a significant impact on solving time. Techniques from formal methods will be used to provide machine assistance for MiniZinc, which is the high-level modelling language to model CSPs. The verification tool, Isabelle, will be used to verify the correctness of reformulations. We plan to apply recent results in formal methods such as program analysis and synthesis to provide semi-automated frameworks for model analysis. We identify the challenges, implement frameworks, and evaluate our experimental results in reformulations for future research.

1.2.1 Motivation

Model reformulation [139] plays an important role in improving models and reducing search space so that solutions can be found faster. In solving CSPs [62], a model of a CSP may be solved rapidly, while a different model may take excessively long to solve. The efficient solution of CSP is significant in real-world applications, such as air traffic management, resource allocation [103], production scheduling, and bioinformatics [11]. Because such problems are often NP-hard, all known algorithms have exponential-time worst-case behaviour. Many technologies such as CP, MIP [85], CBLS, satisfiability modulo theories (SMT) [14,110], and SAT [14] could have different solvers and backends to solve the real-time problems. Each technol-
1.2. Research methodology

tology has its scope of application, and none of the technology is dominant all problems. CP [128] is pervasive and widely used to solve real-time problems which input data could be scaled up to the enormous sizes, and results are required to be given efficiently and dynamically.

CSPs can be reformulated automatically or semi-automatically. The automatic reformulation of CSPs can be obtained using various methodologies and approaches [20, 27, 30, 100, 101, 102]. The automation in modelling has been advanced in five areas such as (1) generating a kernel, in which the compilation in a direct or non-direct system, (2) identifying symmetries, (3) breaking symmetries, (4) adding implied constraints, and (5) transforming constraints. These five advances are discussed in an invited talk at the 10th international workshop on constraint modelling and reformulation [63]. In addition, the reformulation may be achieved semi-automatically [4, 35, 36]. One approach is using a learning-based reformulation approach [164] or improving and stepping towards the automatic methods work [136].

The reformulation of a model into a logically equivalent model is a valuable tool to assist a modeller. A correct reformulation can significantly improve the solving time with the confidence of modeller. The Global Constraint Catalogue [16], the global-constraint library of the MiniZinc system [108], and the Essence system [64, 111] are a few resources for reformulation. Both MiniZinc [108] and Essence [64, 111] are solver-independent languages for modelling CSPs with many common global constraints. In addition, their reformulations are used when a target solver does not have the required globals. Reformulations are inferred in both the Model Seeker [18] and the Globalizer [96] by testing many combinations of global constraints on possible input data and then ranking the possible reformulations.

No proofs of correctness are provided for these reformulations, and the modeller must decide the correctness of the suggested reformulations. No work has been done on verifying the correctness of such reformulation rules, although there are tools [64, 108, 111] that could generate reformulation rules.

1.2.2 Objectives

Interactive theorem provers are notoriously difficult to use. Moreover, the development of proof tactics can be difficult in a new application area such as CP. However, even suggesting and verifying only a few reformulation rules will remarkably benefit modellers of CP and will still be a major advance over state of the art. Machine assistance that provides verified reformulation rules allows modellers to improve their models by reformulations with confidence. The contributions of the Ph.D. project are as follows

- Develop general-purpose tactics for reformulations of models in CP.
• Propose proof systems to prove soundness and completeness of reformulation rules.

• Provide semi-automated and automated frameworks for model analysis, synthesis, and refinement.

• Implement a library of automatic generation of verified reformulation rules for the MiniZinc toolchain.

The key objectives of the Ph.D. project are to support modellers to generate equivalent models that are verified and solve CSPs faster. The novelty and originality of the project are that although many constraint modelling toolchains support to generate reformulation rules [64,108,111], there is not any work that has been done on verifying the correctness of such generated reformulation rules. For instance, with possible input date, the Model Seeker [18] and the Globalizer [96] test many combinations of global constraints. Then, all possible reformulations are ranked and finally provided to modellers without verifying their soundness and completeness. Applying theorem proving techniques and verifying reformulation rules crucially differentiate our work from other existing works. Even though just a few generated reformulation rules is verified, it could be a significant advance to non-expert modellers over state of the art.

When writing this thesis, it does not exist any model reformulation system with the auto-generated proof of models equivalence. The problem consists of several components and research questions that are needed to solve gradually and incrementally. It requires comprehensive knowledge, which is related to programming languages, theorem proving techniques, optimisation methods, and machine learning. In order to solve the problem, we divide the problems into sub-problems with corresponding questions. For instances, (1) how to find the implied constraints in a given model? (2) how to assert that the improved model with supplemental implied constraints is better than the original model? (3) how to automatically derive the formulas of basic and improved models? (4) what and how the translation between modelling language and proof language? (5) how to develop an integrated system that takes the original model, produces an improved model with the proof of equivalence between these models?

To the engineering aspects, the system should contain four components as follows

• Preprocessing component: to transform the model to corresponding theorem prover input and vice versa.

• Reformulation component: to produce an improved model from a given model.
1.3 Contributions

- Theorem prover component: to prove the soundness and completeness of two models.

- Intermediate language component: to translate between modelling language and theorem proving representation.

1.2.3 Methodology

In the scope of the project, we aim to answer all questions as mentioned above. In the initial phase, we firstly focus on solving the problem manually. In the second phase, we selectively improve some manual components to semi-automatic components. In the third phase, we tackle the automated-components problem. Finally, we accomplish the project by integrating and benchmarking the system.

In the first phase, MiniZinc’s reformulation rules will be formalised in Isabelle [115] or a similar system. We take advantage of Isabelle theorem prover to prove the soundness and completeness of two models. We aim to develop general-purpose tactics for the reformulation of models.

We investigate more specialised theories that will be useful in proving the correctness and soundness in the second phase. We will investigate logics for proving the correctness and soundness of reformulation rules and their integration into MiniZinc.

Finally, we will benchmark the system using several case studies that could be found from MiniZinc competition [144], and CSPLib [138].

Our preliminary results are summarised as follows

- a survey of model reformulations that compares, contrasts, and systematically categorises several approaches

- a publication which aims to solve a typical CSP using different solvers such as Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling over two versions of MiniZinc toolchain.

1.3 Contributions

Our contributions are summarised by presenting the overview of projects and contributions. Since we have been working on different projects, consequently, the contributions are vary. There are three grounds of contributions, the first two grounds are related to programming language optimisation and a short-term project. The first ground is related to our previous Encore project, which is about developing a research programming language, and optimise the language construct. Our contribution is received the best
paper [155]. The second ground is about the course project, which are published in [154]. Because the two grounds are main focus in the context of the main theme of the thesis, they are not described in details.

Firstly, in [128], there is a section about reformulations of CSPs in 2006. To the best of our knowledge, there is not adequate and comprehensive survey about model reformalations that compares, contrasts, and systematically categorises several approaches. The contribution is described in Section 3.1.

Secondly, we propose solutions to solve a typical CSP using different solvers such as Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling over two versions of MiniZinc toolchain [152,153] in Section 3.2.

Thirdly, we develop a reformulation process for counter automata in MiniZinc which is presented in Section 3.3. We propose a compiler that takes the input specification and translate to corresponding output in MiniZinc code that supports automata with counter. Finally, we present our experiments with different specifications, and analyse the results as well as the limitations.

Finally, in Section 3.4 we plan to apply recent results in formal methods such as program analysis and synthesis to provide semi-automated frameworks for model analysis. We identify the challenges, implement frameworks, and evaluate our experimental results in reformulations for future research [151].
Chapter 2

Reformulations

2.1 Constraint Programming

In this section, we describe the preliminary concepts such as CP, CSPs, reformulations of CSPs, and modelling languages. The section is partly based on [73, 74, 114, 139, 145].

CP is a programming paradigm for solving combinatorial and optimisation problems. In CP, relations between variables are stated in the form of constraints making CP differ from the common primitives of imperative programming languages. A constraint is a relation that defines valid values for a given set of variables. CP is a form of declarative programming in the sense of that no step or sequence of steps are specified for executing, and the properties of a solution to be found via constraints. Constraints on the feasible solutions for a set of decision variables are declaratively stated by a constraint programmer. CP is categorised into two types, related to mathematical programming and computer programming, respectively [62]. Since the user declaratively states constraints for the finite set of decision variables, CP is partly in the sense of programming in mathematical programming. On the contrary, a search strategy might need to be programmed additionally by a constraint programmer.

There have been several real-world application areas using CP such as packing, airspace sectorisation, robotic task sequencing, doctor rostering, inference of haplotypes, scheduling and planning, vehicle routing, configuration, power and/or oil networks, and bioinformatics. In [128], the authors discuss the historical foundations and an overview of all aspects of CP. A more pedagogic approach to CP appears in [9, 46, 98].
2.2 Constraint Satisfaction Problems

A CSP is a combinatorial problem that is modelled as a finite set of variables, representing the objects the problem deals with, and a finite set of constraints, representing the relationships among the objects. A formal definition of CSP can be found in [62]. In brief, a CSP is a triple of $<X, D, C>$ where $X = x_1, ..., x_n$ is a set of variables, $D = D(x_1), ..., D(x_n)$ is a set of domains containing the values that each variable may take, and $C = c_1, ..., c_m$ is a set of constraints. Each constraint $C_i = <S_i, R_i>$ is a pair of the constraint scope $S_i$ and the constraint relation $R_i$. In other words, $S_i$ is a list of variables, and $R_i$ is a subset of the Cartesian product of domains of the variables in $S_i$. An assignment is a pair $(x_i, a)$ that means the variable $x_i$ in $X$ is assigned the value $a$ in $D_i$. A compound assignment is a set of assignments to distinct variables in $X$. A complete assignment is a compound assignment to all variables in $X$. Valid assignments to the variables in their scope are specified by the relation of a constraint $c = <S_c, R_c>$ where constraint scope $S_c = x_{i1}, ..., x_{ik}$ and value $<a_1, ..., a_k>$ in $R_c$. This means the compound assignments that assign $a_i$ to $x_{ik}$ are valid assignments and satisfy the constraint $c$. An assignment is valid if the value given to each variable is within its domain bounds.

A solution to a CSP is an assignment such that for every constraint $c$ in $C$, the restriction of the assignment to the scope $S_c$ is satisfied. A tuple $A = <a_1, ..., a_n>$ is a solution to the CSP where $a_i$ in $D_i$, and each $C_j$ is satisfied in that $R_{S_j}$ holds on the projection of $A$ onto the scope $S_j$. The goal is to find the set of all solutions to determine whether the set of solution is empty or just to find any solution. If the set of solution is empty, it means the CSP is unsatisfiable. The classic CSP paradigm can be specialised with respect to domains and constraints intensionally or extensionally. A relation $R_i$ may be represented intentionally in terms of an expression that states the relationship that must hold amongst the assignments to the variables it constraints, or it may be specified extensionally by listing its acceptable tuples.

To precisely define the CSP representing a problem $P$, it is difficult. In modelling a problem as a CSP in practice, this definition is not chosen. Instead, variables and values are chosen to represent entities in problem $P$ and the constraints on these variables are written to represent the rules and restrictions defining the solutions problem $P$. The ultimate goal in choosing a model $M$ of a problem $P$ is solve the problem quickly and effectively.

To illustrate the CSP, constraints, and model of CSP, let us consider the following example of a nurse rostering problem.
2.2. Constraint Satisfaction Problems

Table 2.1: Nurse Rostering Problem.

<table>
<thead>
<tr>
<th>Nurse A</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Nurse B</td>
<td>app</td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
<td>call</td>
</tr>
<tr>
<td>Nurse C</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>app</td>
<td>app</td>
<td>call</td>
<td>none</td>
</tr>
<tr>
<td>Nurse D</td>
<td>app</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Nurse E</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Example: Nurse Rostering Problem [120,129] Consider the example of nurse rostering problem [103]. A set of nurses needs to be rostered from Monday to Sunday, and a set of constraints are to be satisfied, namely there is at least one nurse-on-call every day, there are no more than two operations per workday, and there are at least 7 operations per week, and so on. The objective function is to either minimise the cost of hiring nurses or satisfy all the given constraints. The Nurse Rostering Problem is given in Figure 2.1.

In the nurse rostering problem, we model the problem using the Mini-Zinc-like language. Here is the model of nurse rostering problem, which expresses parameters, decision variables, constraints, and objective function. In this problem, the schedule must satisfy constraints such as the number of nurses-on-call per day = 1 (line 9). For each nurse, the number of operations per workday ≤ 2 (line 10), the number of operations per week ≥ 7 (line 11), the number of appointments per week ≥ 4 (line 12), a day off after operation day. Depending on the specific problem specification, an objective function could be plugged in (line 8) and other constraints could be added (line 14). The regular constraint (line 13) is used to enforce that a sequence of variables take a value defined by a finite automaton [107].

```plaintext
1 set of int: Days = 1..7;
2 set of int: Mon2Fri = 1..5;
3 enum Nurses = {Nu_A, Nu_B, Nu_C, Nu_D, Nu_E};
4 enum ShiftTypes = {app, call, oper, none};
5
6 array[Nurses,Days] of var ShiftTypes: Roster;
7
8 solve minimize ...; % plug in an objective function
9 constraint forall(d in Days)(count(Roster[.,d],call) = 1);
10 constraint forall(w in Mon2Fri)(count(Roster[.,w],oper)<=2);
11 constraint count(Roster,oper) >= 7;
12 constraint count(Roster,app) >= 4;
13 constraint forall(D in Nurses)(regular(Roster[D,..],
   (oper|none|app|call));
```
2.3 Constraint Modelling

In practice, the problems are specified in natural statements. The specifications of problems are later constructed and are modelled using modelling languages. CP systems implement variables and constraints and provide solutions for the problems that satisfy all the constraints. Figure 2.1 illustrates the whole process.

Figure 2.2 contrasts the paradigms of modelling and programming approaches [118]. Solutions in a modelling approach may be obtained automatically from models, which are declaratively derived from specifications. In contrast, solutions in a programming approach may be achieved using algorithms implemented manually.

The general process of modelling and solving problems is using one modelling language to express models, and pipelining the models to different solver languages. The general pipelining process is illustrated in Figure 2.3.

In the pipelining process, the models may be compiled to multiple solvers. Figure 2.4 illustrates the general pipelining process starting from using modelling language to implementing the model and ending with common low-level modelling language, which is compatible with many different solver languages.

Example of ESSENCE Pipelining process: The models are expressed in ESSENCE language, and later compiled in ESSENCE’ language, which can be used as input for a variety solvers such as ECLiPSe, Minion, Gecode,
2.4 Modelling Languages

In this section, we describe the modelling languages using a high-level problem specification language to specify a concrete CP model. The rest of this section presents the most important toolchain and modelling languages.

**MiniZinc** [108] is a free and open-source constraint modelling language to model CSPs and optimisation problems in a high-level, solver-independent way. *MiniZinc* aims to be a standard language for modelling CSPs with or and FlatZinc. In the ESSENCE pipelining process, there are two different inputs to start the process, which are ESSENCE’ and XCSP. The process ends with the Flat ESSENCE’ that is the input to many different solvers. Figure 2.5 shows the pipelining process to multi-solvers of ESSENCE.

**Example of MiniZinc Pipelining process** Figure 2.6 illustrates the *MiniZinc* Pipelining process. Models in *MiniZinc* are compiled into FlatZinc language, which is compatible with many other solvers, e.g., G12FD, SAT, SMT, Gecode, Chuffed, and Gurobi.

![Figure 2.2: Modelling and programming approaches.](image)

![Figure 2.3: General Pipelining Process to one-solver.](image)
without optimisation over Booleans, integers, real numbers, enumeration, multidimensional array of elements, set of elements, and recently Strings. It is a declarative constraint modelling language and supports annotation for specifying choices such as search strategies. One of the interesting features of MiniZinc toolchain is that the MiniZinc model can be easily mapped onto various solvers by compiling its models and data files into FlatZinc instances.

FlatZinc is a low-level solver input language. The FlatZinc models are front-end inputs for several existing solvers and backends such as Gecode [69], ECLiPSe [10], SICStus Prolog [29], and a number of solvers, e.g., Zinc [44,97] and Mercury [15], developed by the G12 research team [142]. The backend with solvers takes the FlatZinc model to give the (optimal) solution. MiniZinc toolchain is illustrated in Figure 2.7 [119].

Example: Job-shop problem [49,59,165,166] MiniZinc instance of the well-known Job-shop problem, where the aim is to schedule \( n \) jobs of different durations, using \( m \) identical machines, running only one job per
2.4. Modelling Languages

Figure 2.6: *MiniZinc* Pipelining Process to multi-solvers.

Figure 2.7: MiniZinc toolchain.

Machine at the same time and minimising the total length of the schedule. In this toy instance, 2 jobs and 2 machines are used and the translation process of *MiniZinc* to FlatZinc is illustrated using this example later on. This example is directly from [114].

The data instance for the model can be defined separately in another file using the *MiniZinc* syntax

\[
\text{size} = 2; \ d = [[2,5] [3,4]]; \\
\]

The Job-shop problem is modelled as follows in *MiniZinc* language.

\[
\begin{align*}
\text{int: size; } & \% \text{ size of problem, number of jobs and machines} \\
\text{array [1..size,1..size] of int: } & \text{d; } \% \text{ task durations} \\
\text{int: total = sum(i,j in 1..size) (d[i,j]); } & \% \text{ total duration} \\
\text{array [1..size,1..size] of var 0..total: } & \text{s; } \% \text{ start times} \\
\text{var 0..total: end; } & \% \text{ total end time}
\end{align*}
\]
predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \ / s2 + d2 <= s1;
constraint
forall(i in 1..size) {
    forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\ 
    s[i,size] + d[i,size] <= end /\ 
    forall(j,k in 1..size where j < k) ( 
        no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
    )
};
solve minimize end;

Compiling the MiniZinc models produces FlatZinc instances, which are a
list of variables declarations, flat constraints (without forall, exists
and list comprehensions), together with or without a variable to optimise.

The translation from MiniZinc models to FlatZinc models consists of two
parts, which are flattening and post-flattening. The details of the translation
are found at [108]. In the flattening part, several reductions such as built-ins
evaluation, comprehension unrolling, compound built-in unrolling, fixed ar-
ray accesses replacement, if-then-else evaluation, and predicate inlining are
iteratively applied until the fix-point is reached. In the post-flattening part,
the following steps are applied in the given order such as stand-alone assign-
ment removal, let floating, Boolean/numeric/set decomposition, (in)equality
normalisation, array simplification, anonymous variable naming, conversion
to FlatZinc built-ins, and top-level conjunction splitting. Since the paradigm
of MiniZinc toolchain is pipelining process to multi-solvers as illustrated in
Figure 2.4, the FlatZinc models could be solved by different solvers such as
Gecode or Chuffed.

Before MiniZinc is introduced, there are some historical languages, and
some of them no longer exist. Since this thesis on the reformulation, we
could name some of modelling languages in the past such as OPL [148],
ESRA [56], NP-Spec [26], F and Fiona [81], CGRASS [67], TAILOR and
SAVILE ROW [126, 44, 97, 143], and CONJURE 1.0 [4].

Other modelling languages

There are more modelling languages other than MiniZinc such as ESSENCE
and ESSENCE’ [64], AIMMS [21], GAMS [22], AMPL [58], Mosel-Xpress
[37], SMT-lib [24], and the like. In addition, several toolchains and backends
exist, for instances, Cadmium, Minion, ILOG, ECLiPSe, Choco, SICStus
Prolog, Gecode, HAMPI, KALUZA, SUSHI, and so on. Based on the fea-
tures of the modelling language that support string constraints, and specify
unknown length, there are several toolchains, extensions, and solvers such as Gecode+S, MiniZinc, Hampi, Kaluza, and Sushi. The Gecode+S usually shows better empirical results than other dedicated string solvers. ESSENSE is a system that is similar to MiniZinc, with flattening and post-flattening parts. In the flattening part ESSENCE translates input models to flattening model and choosing specific solvers to solve the flattening models in the post-flattening part. The difference between ESSENCE and MiniZinc is that the ESSENCE pipelining process is capable for two kinds of inputs to start the process.

2.5 Reformulations of CSPs

With a given model $M$ of a problem $P$, there are several ways to improve the performance of model $M$ by, for instance, changing viewpoint, expressing the constraints differently, using auxiliary variables, finding implied constraints or reformulating the CSPs. In order to express the constraints differently, one can choose to combine the constraints, eliminate variables, use global constraints, add external constraints, or find reified constraints and meta-constraints. The reformulation of CSPs could be done at high level [80]. A global constraint is a constraint that can be over arbitrary subsets of the variables. A reified constraint, also known as meta constraint, $c \leftrightarrow b$ reflects the truth value of the constraint $c$ onto a 0/1-variable $b$, which means $b$ takes the value 1 if the constraint $c$ is satisfied and 0 otherwise.

One approach to systematically categorise the reformulations of CSPs is based on the evolution of the model. A model $M$ of problem $P$ may be evolved in several ways, and the reformulations of CSPs can be grouped into three major types as follows. The first type is that a model is reformulated to model’, within the same modelling language [139,146]. The second type is that a model with non-CP standard types is compiled to another model with CP standard types [74]. The third type is that a model is converted to SAT, MIP, SMT, or others [52,74,109,114].

The reformulation may be obtained using different approaches such as automation, semi-automation, or manual reformulations in general, and MIP to CP, implied constraints, set CSP to CSPs, String variables to CSP without String variables, symmetry breaking, precomputation, non-binary to binary translations, or other approaches in particular. In the Handbook of CP [139], there are several approaches to model reformulation, e.g., changing viewpoint, expressing the constraints by combining constraints, eliminating variables, adopting global constraints and/or external constraints and/or reified constraints and meta-constraints, and auxiliary variables.

Implied constraint technique is to add more constraints that may reduce
the amount of search and search space without changing the set of solutions. **Example: Car sequencing problem** [137]: A number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope with. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option. The problem has been shown to be NP-complete [70]. To reduce the search effort and run-time, useful implied constraints are added such as (1) existing constraints only say that the option capacities cannot be exceeded, but we can not go too far below capacity either, (2) suppose there are 30 cars, and 12 require option 1 (capacity 1 car in 2), (3) at least one car in slots 1 to 8 of the production sequence must have option 1, otherwise 12 of cars 9 to 30 will require option 1, i.e. too many, (4) cars 1 to 10 must include at least two option 1 cars,...; and cars 1 to 28 must include at least 11 option 1 cars.

**Symmetry breaking technique** is to break symmetry in matrix modelling. The additional symmetry-breaking constraints reduce the set of solutions by eliminating symmetric ones, which improve the search by avoiding symmetric branches. For example, recall the **Nurse rostering problem** [2.2] the nurses or rows can be permuted, 5! variable symmetries.

**Multiple viewpoints technique** is to model the problem from different perspectives. A problem can be modelled using different viewpoints, which have both benefit and drawbacks. Multiple viewpoints can be combined together with the use of **channelling technique**.

**Channelling technique** is to combine the best constraint formulation from each model, and connect the sets of variables from each model through channelling constraints.

- General Reformulations (Section 2.5.1).
  - Type 1 With the same language, a model is reformulated to model’.
  - Type 2 A model (with non-CP standard types) is compiled to another model (with CP standard types).
  - Type 3 A model is reformulated to SAT/MIP/SMT/others.

Integer Programming (IP) is an optimisation program that specifies only integer variables, a set of linear equality and equality constraints, and only
for optimisation problems. Mixed Integer Programming (MIP) is linear equalities and/or inequalities over floating-point and integer variables. MIP models are notoriously difficult to formulate [162], but for some class of combinatorial problems such as an employee timetabling problem, a new MIP modelling approach can significantly decrease computational times in comparison with a classical MIP formulation [38]. MIP is applicable for many classes of combinatorial problems, which can be subdivided into sequencing problems and allocation problems [19, 43, 89, 93, 147, 149, 160, 161].

This section systematically categorises various approaches in model reformulation and describes these approaches in detail. We categorise the model reformulations into three general types. In the next sections, we unify the three general reformulations and organise model reformulations in different approaches. Recall to Section 2.5, the reformulations of CSPs could be categorised into several categories and types based on a variety of approaches. The categories of reformulations are based on general reformulations of a model, reformulation by levels of automation, MIP to CP, set CSPs to CSPs, string variables to CSP without String variables, symmetry breaking, precomputation, non-binary to binary, and other approaches.

2.5.1 General Reformulations

One approach to systematically categorise the reformulations of CSPs is based on the evolution of the model. A model of a problem may be evolved in several ways, and the reformulations of CSPs can be grouped into three major types as follows.

**Type 1** A model is reformulated to model’, within the same modelling language [139, 146], which is illustrated in Figure 2.8. This approach is general in the modelling process and produces a good model, which leads to an efficient resolution of a given problem. In the first type, all principles of modelling such as symmetries breaking, implicit constraints, global constraints, redundant constraints, and dominance rules can be applied [125].
A model with non-CP standard types is compiled to another model with CP standard types, which is demonstrated in Figure 2.9. In Chapter 17 of the Handbook of CP [74], the authors present higher-level modelling facilities utilising constraints over structured domains. The authors introduce a number of significant research topics and available results in software and systems that embed constraints over structured domains. The major approaches such as constraints over regular and constructed sets, which embed strings and constructed sets in CP, and constraints over finite set intervals are reviewed.

One of the approaches coming along with this type is to support non-standard types such as strings constraints. String constraints are not natively and practically supported by many solvers. In [8], the authors introduce the extension of the MiniZinc that can specify string variables of unknown length. A new extension, called Gecode+S [132], which is an official part of Gecode, with bounded-length string variables of Gecode solver [69] also supports string constraints [41, 42].

A model is reformulated into SAT, MIP, MILP, or SMT, which is represented in Figure 2.10. In a solving technologies context, with general-purpose solvers that take models as input, there are a lot of solving technologies, for instance SAT, SMT, IP and MIP, and CP. In addition, several technologies are combined together, called hybrid technologies, such as Lazy clause generation (LCG) that uses CP propagators to generate clauses in a SAT solver, large neighbourhood search (LNS) that follows a local search (LS) procedure with additional steps to find an (optimal) solution to the subproblem, constrained integer programming (CIP) [55]. Moreover, two encodings of binary CSPs are examined into SAT in [52]. The advantage of their results is to reduce both the runtime and the number of search nodes used by a SAT solver on the encoded CSPs. In general, the combinatorial problems can be represented as CSPs, and there exist a wide range of techniques to deal with CSPs by modelling the problem as graphs [74]. One can model the CSPs as a propositional satisfiability formula, which can be solved by SAT solvers [109]. In [114], the authors develop two systems to reformulate CSPs into SMT instances, which can be solved using SMT.
2.5. Reformulations of CSPs

Figure 2.10: A model is reformulated to SAT, MIP, or others.

solvers. The reformulation into SAT, MILP, and SMT formalisms is briefly introduced in [114]. CSPs can be solved in a different way by reformulating them into other common formalisms such as SAT, MILP, and SMT.

Tailor [126] shows another approach related to reformulation. The approach performs highly-effective and efficiency-enhancing transformations in several steps. The transformations can be done by replacing common subexpression by a new variable, removing duplicate constraints, reformulating to increase the number of applications, quantifying optimisations before unrolling, e.g., moving invariant expression outside the scope of quantifiers.

2.5.2 Reformulations by Levels of Automation

One approach is to categorise as the reformulations of CSPs based on the level of automation. CSPs can be reformulated automatically in several different ways. In this category, we focus on the work that reformulates and transforms the CSPs automatically; however, these categories can also be categorised with various perspectives. In [52], the authors propose two automatic encodings to reformulate binary CSPs into propositional satisfiability (SAT) that reduce both runtime and the number of search nodes. In [13], the authors describe a prototype modelling system that automatically refines an abstract specification of a CSP into a set of alternative constraint programs [20, 27, 100]. By exploiting dominance relations in constraint optimisation problems to dramatically reduce the search space, [101, 102] propose an automatic method to detect some of the dominance relations manually identified in [34]. In [94], the authors propose an algorithm to automatically convert ordinary table constraints into compact smart table constraints, which represent compactly a number of well-known global constraints and arbitrarily structured constraints. In [141], the authors present a new approach for automatic generation and selection of streamlined constraints models via Monte Carlo Search on a model lattice. They expand upon the method in [159], which generates effective streamliners automatically from the specification of ESSENCE [65, 66]. The automation in modelling has been advanced in five areas (1) generating a kernel, (2) identifying symmetries, (3) breaking symmetries, (4) adding implied constraints, and (5) transforming constraints. These five advances are discussed in an invited
talk at the 10th international workshop on constraint modelling and reformulation [63]. In [4], the authors present the CONJURE 1.0 automated constraint modelling system to reproduce the kernels of the constraint models. All ESSENCE specifications are able to be refined by CONJURE 1.0.

**Semi-automatic reformulation**

In addition, the reformulation may be achieved semi-automatically. The idea of automatically learning constraint networks framework, called CONACQ, is introduced in [35, 36]. The CONACQ framework provides semi-automatic methods for acquiring constraints to assist the human user. Another semi-automatic learning-based reformulation approach is introduced in [164]. Their contributions are based on the work in [136] that improve and step towards the automatic methods. The automation phase is to rename the literals in learned clauses to make it easier for modellers to understand the clauses.

### 2.5.3 Reformulation by Changing Viewpoint

Reformations by changing from one viewpoint to another are standard and useful for specific problem classes. Changing viewpoint may require literally studying the problem from a different angle, and developing some understanding of the problem [6, 83, 104, 140]. In [33], the authors give general theorems for proving propagation redundancy of one constraint with respect to channelling constraints and constraints in the other model. They define a broad form of channelling constraints that are covered by their approach. The authors use problems from CSPLib\(^1\) to illustrate how detecting and removing the propagation of redundant constraints can significantly speed up solvers. In [99], the authors present an algorithm that produces correct channelling constraints for the generated models using only the facilities already provided by the CONJURE system. It is the automatic modelling tool to generate CSP models from problem specifications [31].

**Example: Objects, Shapes, and Colours problem** \([28, 92, 119]\) As a motivating example, we consider the *Objects, Shapes, and Colours* problem to illustrate the changing viewpoint approach. There are \(n\) objects, \(s\) shapes, and \(c\) colours, with \(s \geq n\). The task is to assign a shape and a colour to each object such that the objects have distinct shapes, the numbers of objects of the used colours are distinct, and other constraints, resulting in NP-hardness, are satisfied. This problem can be modelled from different viewpoints. The following models, which are directly from [119] illustrate three different viewpoints.

\(^1\)http://www.csplib.org/
We illustrate viewpoint 1, which colours, if any, does each shape have? In this model, an array `Colour` of decision variables is used to represent which colour each shape has (line 8).

```plaintext
% Viewpoint 1: Which colour, if any, does each shape have?
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");
array[1..s] of var 0..c: Colour; % 0 is a dummy colour
constraint count(Colour,0) = s-n;
% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(global_cardinality(Colour,1..c));
% The objects have distinct shapes:
% implied by lines 6 and 9
% ... add here the other constraints ...
solve satisfy;
```

We illustrate viewpoint 2, which shapes, if any, does each colour have? In this model, an array `Shape` of decision variables is used to represent which shape each colour has (line 8).

```plaintext
% Viewpoint 2: Which shapes, if any, does each colour have?
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");
array[1..c] of var set of 1..s: Shapes;
constraint n = sum(colour in 1..c)(card(Shapes[colour]));
% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(colour in 1..c)
    (card(Shapes[colour]));
% The objects have distinct shapes:
constraint n = card(array_union(Shapes));
% ... add here the other constraints ...
solve satisfy;
```
We illustrate viewpoint 3, which shape and colour does each object have? This model uses both Shape and Colour arrays of decision variables.

```plaintext
% Viewpoint 3: Which shape & colour does each object have?

int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");
array[1..n] of var 1..s: Shape;
array[1..n] of var 1..c: Colour;

% There are n objects: implied by lines 8 and 9
% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(global_cardinality(Colour,1..c));
% The objects have distinct shapes:
constraint alldifferent(Shape);
% ... add here the other constraints ...
solve satisfy;
```

### 2.5.4 Reformulate Implied Constraints

Implied constraints or redundant constraints are constraints, which are implied by the constraints defining the problem [139]. The property of implied or redundant constraints is that the set of solutions is not changed by implied and or redundant constraints. As a result, they are logically redundant. However, implied or redundant constraints are useful for reducing the search effort to solve CSPs. Implied constraints are widely used, for instance, [50] uses implied constraints in solving the car sequencing problem. The relationship between implied constraints and search order, implied constraints and global constraints, implied constraints from subproblems are described and discussed in detail in [139]. In [61], the authors construct implied constraints for automaton constraints based on their earlier work [60,82], where the constraints implied by the decomposition are added to improve the propagation. They present a fully automated parametric tool that selects, in an off-line process, a set of non-redundant linear constraints that are implied by decomposition in [17] of a constraint on a sequence of variables.

**Example: Magic Series problem [156]** To illustrate the reformulation using implied constraints, we consider the Magic Series problem as an ex-
ample. The element at index \( i \) in \( I = 0...(n-1) \) is the number of occurrences of \( i \), e.g. for \( n = 4 \), the solution is \( \text{Magic} = [1, 2, 1, 0] \). The domain of each variable of the \( \text{Magic} \) array is \([0...n]\). Here is the constraint which is directly from [119].

\[
\text{global_cardinality_closed}(\text{Magic},I,\text{Magic})
\]

The problem can be reformulated using the implied constraint. More specifically, by using a CP solver for \( n = 80 \), only 7 search nodes are explored instead of 302, and the solving speed is 1,000 times faster than without using the implied constraint.

\[
\text{sum}(\text{Magic})=n \setminus \text{sum}(i \ in \ I)(\text{Magic}[i]*i)=n
\]

### 2.5.5 Reformulation by Symmetry Breaking

A key problem in CP has long been recognised: search can revisit equivalent states over and over again. Symmetry has become a major research area. Briefly, breaking symmetries can be applied by adding constraints before a search, using different dynamic symmetry methods, combining symmetries breaking methods, and expressing and detecting symmetry. The details of many symmetry exclusion methods are described in Chapter 10 of the Handbook of CP [73]. Symmetry breaking plays an important role in speeding up the search in CSPs that contain symmetry [25,40,54,57,117,123,127]. Symmetry occurs in many CSPs, and it must be considered or it will waste much time visiting symmetric solutions. One mechanism to break the symmetry is to add constraints, which eliminate symmetric solutions [122]. Another simple method for breaking any type of symmetry between variables has been presented in [39]. To deal with symmetries, in [158], the authors introduce symmetry breaking constraints that apply to variables and values, conditional symmetries, as well as symmetries working on set and other types of variables. Modelling and reformulation are equally important for symmetry breaking [128]. In extreme cases, reformulation of problems can be critical in dealing with symmetries.

**Example: Social Golfers Problem** [51, 79] This is problem 10 in CSPLib. In this well known problem, \((g \cdot s)\) golfers want to play in \( g \) groups of \( s \) slots each week, so that any two golfers play in the same group at most once, for as many weeks as possible. The difficult case is to find all solutions for \( w \) weeks. The requirement is to find the schedule \( \text{Weeks}(w) \times \text{Groups}(g) \times \text{Slots}(s) \to \text{Players}(g \cdot s) \) subject to the constraints that any two players are at most once in the same group. A solution for \(<w, g, s> = <4, 4, 3>\) is illustrated in Table 2.2.
Table 2.2: A solution for the Social Golfers Problem for \( <w, g, s> = <4, 4, 3> \).

<table>
<thead>
<tr>
<th></th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Group 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>[1,2,3]</td>
<td>[4,5,6]</td>
<td>[7,8,9]</td>
<td>[10,11,12]</td>
</tr>
<tr>
<td>Week 2</td>
<td>[1,4,7]</td>
<td>[2,5,10]</td>
<td>[3,8,11]</td>
<td>[6,9,12]</td>
</tr>
<tr>
<td>Week 3</td>
<td>[1,8,10]</td>
<td>[2,4,12]</td>
<td>[3,5,9]</td>
<td>[6,7,11]</td>
</tr>
<tr>
<td>Week 4</td>
<td>[1,9,11]</td>
<td>[2,6,8]</td>
<td>[3,4,10]</td>
<td>[5,7,12]</td>
</tr>
</tbody>
</table>

Table 2.3: A symmetry breaking solution for the Social Golfers Problem for \( <w, g, s> = <4, 4, 3> \).

<table>
<thead>
<tr>
<th></th>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Group 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>[1,2,3]</td>
<td>[4,5,6]</td>
<td>[7,8,9]</td>
<td>[10,11,12]</td>
</tr>
<tr>
<td>Week 2</td>
<td>[1,4,7]</td>
<td>[2,5,10]</td>
<td>[3,8,11]</td>
<td>[6,9,12]</td>
</tr>
<tr>
<td>Week 3</td>
<td>[1,8,10]</td>
<td>[2,4,12]</td>
<td>[3,5,9]</td>
<td>[6,7,11]</td>
</tr>
<tr>
<td>Week 4</td>
<td>[1,9,11]</td>
<td>[2,6,8]</td>
<td>[3,4,10]</td>
<td>[5,7,12]</td>
</tr>
</tbody>
</table>

By observing Table 2.2 the weeks, groups, group slots, and players of a social golfer schedule are not distinguished. The weeks/rows can be permuted: there are 4! variable symmetries. The groups can be permuted within a week: there are 4!\(^4\) variable symmetries. The group slots can be permuted: there are 3!\(^6\) variable symmetries. The player names can be permuted: there are 12! value symmetries. To perform symmetry breaking by reformulation, we break the slot symmetries within each group by switching from the 3D \( x \times g \times s \) array of integer variables in Table 2.2 to a 2D \( w \times g \) array of set variables as in Table 2.3 and adding the constraint that all sets must be of size \( s \).

### 2.5.6 Reformulation by Precomputation

Precomputation and/or presolving is a well-known concept in MIP, and SAT modulo theories to facilitate solvers to transform models. In 2016, the reformulation by precomputation approach is introduced in [47]. Their study is extended in another publication in 2017 using auto-tabling, which replaces model parts by constraints with precomputed solution arrays [48]. The authors propose an automatic process for tabling and integrate it in the MiniZinc toolchain that enables auto-tabling feature by annotating the predicate definitions to the table. In this research mainstream approach, several add-ons, extensions, and libraries are implemented related to auto-tabling. In the Propia library [95] of ECLiPSe, a table constraint internally replaces all precomputed solutions, which are annotated appropriately. With IBM ILOG CPLEX CP Optimizer, a corresponding table constraint is constructed as an implied constraint based on finding of all solutions to the constraints involving a set of variables [87]. Another approach related to reformulation by precomputation is to speed up the solving process by transforming a part of a model such as CHR rules [1], stateless C-code [72], multivalued decision
Example: Prize-pool Division problem \[119\] To illustrate the reformulation by precomputation, we consider the example of a Prize-pool Division problem. Consider a maximisation problem where the objective function is the division of an unknown prize pool by an unknown number of winners. A part of the model of the prize-pool division problem is illustrated in MiniZinc-like language.

```plaintext
array[1..5] of int: Pools = [1000, 5000, 15000, 20000, 25000];
var 1..5: x;
var 1..500: numWinners;
solve maximize Pools[x] div numWinners;
```

By observation, the `div` function on decision variables should be avoided because it yields weak inference, at least in CP and LCG solvers. The inference takes unnecessary time and memory, and it is not supported by all MiniZinc backends. The idea is to pre-compute all possible objective values. For each possible value pair of \(x\) and \(\text{numWinnders}\), we pre-compute a 2D array, which is indexed by 1..5 and 1..500.

```plaintext
array[1..5] of int: Pools = [1000, 5000, 15000, 20000, 25000];
var 1..5: x;
var 1..500: numWinners;
array[1..5,1..500] of int: objFun = array2d(1..5,1..500, [Pools[p] div n | p in 1..5, n in 1..500]);
solve maximize objFun[x,numWinners];
```

The main difference between two models is that in the pre-computed approach the parameter `Pools` is exploited. This could help a solver to reduce search space and save the solving time.

2.5.7 Reformulate SET-CSPs to CSPs

In \[41\]\[42\], the authors present a decision procedure for sets, binary relations, and partial functions. The authors prove that the procedure is sound, complete and terminating. The Prolog implemention of the procedure is presented. In \[3\]\[78\], the authors introduce set variables and set constraints
in local search by extending relevant local search concepts to simplifying the solving of CSPs. They also present a framework for constraint-based local search for modelling and solving combinatorial problems with set variables and set constraints. In [11], the authors present some applications of a set constraint solver Cardinal. It extends constraint solving to set variables with attached set functions and with special inferences over them. In [106, 130], the authors propose a constraint-based approach for solving set partitioning problems. They show that an efficient and easily modifying model is obtained by using a global constraint propagator to enforce consistency between local knowledge and global knowledge. This propagator can be used to prune the search space efficiently.

2.5.8 Reformulate String Variables

A substantial amount of research in recent years on the development of solvers for string constraints has been done. In [8], the authors define an interpreter for converting a MiniZinc model with strings into a FlatZinc instance. The interpreter allows a modeller to define string variables and a suitable set of string constraints as a built-in feature of MiniZinc and translates the string variables into integer variables [8]. In [2], the authors develop a tool, which is integrated into CEGAR-based model checker for the analysis of programs encoded as Horn clauses. The feature of the tool is the ability of automatically establishing the correctness of several programs. In [68], the authors construct a string constraint solver, called SUSHI, which is constructed for solving Simple Linear String Equation (SISE) constraints. SISE is a decidable fragment of the general string constraint system modelling a collection of regular replacement operations, which are frequently used by text processing programs. Yet another string solver, named HAMPI, is introduced in [91]. HAMPI is a string solver for string constraints over bounded string variables. It can find a bounded string that satisfies all the constraints or exposes the unsatisfiable constraints otherwise. In [84], the authors propose a constraint solving algorithm for equations over string variables coming up with a prototype that constructs the search space lazily based on an automata representation of the constraints [75, 133, 134, 135].

2.5.9 Reformulate Non-binary to binary translations

The reformulation related to non-binary to binary translation arises upon the context that earlier search algorithms for CSPs only deal with binary constraints. Consequently, standard transformations of a CSP with non-binary constraints into a binary CSP are studied [139]. The original constraints are replaced by new variables using the dual graph translation of
a non-binary CSP. The translation produces a new CSP based on a different viewpoint. These transformations are investigated by Bacchus and van Beek [12] using a forward checking algorithm. Both the hidden variable and dual transformation that outperformed the original model are shown. However, since there are better ways of dealing with many types of non-binary constraint, in practice their approach has been little used. In [163], the authors tackle the uncertain CSP and propose two conditions that ensure that a tractable reformulation exists, and give an algorithm to test for the conditions for binary constraints. In [88], the authors introduce the first encoding that is linear in the number of variables, domain size, and constraint size w.r.t. the size of the SAT instance. A new binary CSP encoding for SAT is presented and theoretically compared with other encodings. The idea of the work is to map non-binary constraints to binary ones in order to exploit the advanced features for binary constraints. In [53], the reformulation of a non-binary CSP into an equivalent binary CSP has been done by developing correct-by-construction solvers using proof assistants like Isabelle, or Coq.

2.5.10 Reformulations into SAT

The reformulation can be done by choosing how to encode the variables and their domains and then translating the constraints taking into account this encoding in such a way that the meaning of these constraints is retained. There are three ways to encode variables: normal encoding, log encoding, and regular encoding. In [52], the authors propose two automatic encodings to reformulate binary CSPs into propositional satisfiability (SAT). They show two encodings of binary CSPs into SAT, which are the direct encoding [157], and the supported encoding [71]. They claim that applying their hyper-resolution reduces both the runtime and the number of search nodes. In [23,113], the authors present a method for solving weighted CSPs by translating a shared Binary Decision Diagram into SAT. The experimentations are performed on the WSimply system. The authors claim that the new technique WSimply outperforms some state-of-the-art solvers in most of the studied instances.

Example: SAT problem  Here is an example of a SAT problem.

```plaintext
1 var bool: w, x, y, z;
2
3 constraint (not w \ / not y) \ / (not x \ / y)
4 \ / (not w \ / x \ / not z)
5 \ / (x \ / y \ / z) \ / (w \ / not z);
```

The clauses at line number 3 only include conjunctions (\ /) of clauses. A clause is a disjunction (\/) of literals. A literal is a Boolean variable or
its negation. There is no objective function for SAT problems. A solution for the SAT problem above is $w = false, x = true, y = true,$ and $z = false$.

2.5.11 Reformulations into MILP and SMT

Reformulations into MILP In order to encode CSPs using MILP, arithmetic constraints are translated directly, the linearisation of those constraints may be necessary in case they are non-linear. In addition, those types of constraints that cannot directly be translated into arithmetic expressions are reified. [74,109,114].

Reformulations into SMT Reformulation of CSPs into SMT can be obtained by choosing one or more background theories that extend a propositional SMT formula. In [114], the authors develop two systems to reformulate CSPs into SMT instances, which can be solved using SMT solvers.
Chapter 3

Summary of Contributions

In this chapter, we will summarise our contributions by presenting the overview of projects and contributions. Since we have been working on different projects, consequently, the contributions are vary. There are three grounds of contributions, the first two grounds are related to programming language optimisation and a short-term project. The first ground is related to our previous Encore project, which is about developing a research programming language, and optimise the language construct. Our contribution is received the best paper [155]. The second ground is about the course project, which are published in [154]. Because the two grounds are main focus in the context of the main theme of the thesis, they are not described in details.

Our contributions are described in details in the third ground.

Firstly, in [128], there is a section about reformulations of CSPs in 2006. To the best of our knowledge, there is not adequate and comprehensive survey about model reformalations that compares, contrasts, and systematically categorises several approaches. The contribution is described in Section 3.1.

Secondly, we propose solutions to solve a typical CSP using different solvers such as Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling over two versions of MiniZinc toolchain [152,153] in Section 3.2.

Thirdly, we develop a reformulation process for counter automata in MiniZinc which is presented in Section 3.3. We propose a compiler that takes the input specification and translate to corresponding output in MiniZinc code that supports automata with counter. Finally, we present our experiments with different specifications, and analyse the results as well as the limitations.

Finally, in Section 3.4, we plan to apply recent results in formal methods such as program analysis and synthesis to provide semi-automated frame-
works for model analysis. We identify the challenges, implement frameworks, and evaluate our experimental results in reformulations for future research [151].

3.1 A Survey of Reformulations of Constraint Satisfaction Problems

We present an overview of reformulation focusing on the contributions made in the area of reformulation of CSPs where significant performance improvements have been achieved. According to criteria and types that categorise reformulation, we systematically unify and organise the vast literature. The survey contrasts and compares different categories of reformulation for solving CSPs. We do the in-depth survey based on more than 150 references, and propose 14 types of reformulations together with examples.

3.2 Towards Efficient Solvers for Optimisation Problems

Nowadays, watching videos online is pervasive, especially watching videos from Youtube. When streaming videos from Youtube to a huge amount of people, who could be in the same city or from different continents, minimising the waiting time for all requests from clients are critical. In the context of the Streaming videos problem, the video-serving infrastructure includes remote data centers locating in thousands of kilometers away, cache servers which store copies of popular videos, and endpoints which each of them represents a group of users connecting to the Internet in the same geographical area. The expected solution for the Streaming videos problem is to decide which videos to put in which cache servers. The specification of the problem could be found in detailed at [76], and the data could be found at [77]. MiniZinc [108] is a constraint-based modelling language for satisfaction and optimisation problems such as Streaming videos problem with independent solving technologies which supports for diverse technologies’ solvers for instances CP, CBLS [150], MIP, SAT, and SAT modulo theories (SMT). In this thesis, the bin-packing approach, which is modelled in modelling language MiniZinc, is used to solve the Streaming videos problem in two different ways: use the built-in global constraint bin_packing_load(), and model the problem using a heuristic.

Our contributions are as follows

- The first contribution is to solve the Streaming video problem by modelling the bin packing algorithm using MiniZinc toolchain, which
compiles the model into FlatZinc, which is the input of five different solvers.

- The second contribution is to perform experiments of two approaches over five solvers inside MiniZinc using 2 versions of MiniZinc.

In this section, we present two approaches for the problem. In the first approach, the bin packing load algorithm is re-implemented in the manual model. The second approach uses the `bin_packing_load` constraint from MiniZinc library with search and strategy annotations to reduce the search space. The two approaches are evaluated on five solvers over two MiniZinc versions.

The unique challenges of cache placement for video streaming services compared to web caching are caching strategies. Since video objects usually have a high read-to-update ratio compared to typical web objects, such as web pages. In case of a web cache and a video cache with the same storage space, the first can store a larger number of objects than the latter. Therefore, different caching strategies are used in caches for video distribution mechanisms.

The disadvantage of those backends is the division computation such as `/` and `div`, which can be avoided by putting the division computation in the output phase. The real question here is how can the MiniZinc model be improved to instantiate and give the result for the biggest data instance, `kittens`, whose size is up to 5.4 MB in text format. The Streaming video problem could be modelled by other modelling language and benchmarked with the same data instances to compare the performance and the efficiency with MiniZinc model.

The experiment is done using two different version of MiniZinc, 2.1.7 and 2.2.1 as it is recently released. In the first experiment, all the instances are conducted using MiniZinc 2.1.7. The test results produced by MiniZinc 2.1.7, and MiniZinc 2.2.1 are marked by (*) and (ψ), respectively.

The model is tested using all five instances, with both MiniZinc 2.1.7 and MiniZinc 2.2.1. All the test results are shown in Section 3.1.

In order to run the test in all backends, the final score computation is done at the output phase to avoid the division computations such as `/` and `div` which are not executable in Chuffed and Gecode.

### 3.3 Automata with Counters in MiniZinc

In this section, we present our contribution in specifying automata with counters in MiniZinc. This is one of the two strands that we investigate, which aim to make more significant research contributions in the area of
<table>
<thead>
<tr>
<th>Technology</th>
<th>CP</th>
<th>LCG</th>
<th>MIP</th>
<th>CBLS</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>Gecode</td>
<td>Chuffed</td>
<td>Gurobi</td>
<td>OscaR.cbls</td>
<td>Lingeling</td>
</tr>
<tr>
<td>Backend</td>
<td>Gecode</td>
<td>Chuffed</td>
<td>Gurobi</td>
<td>fzn-oscar-cbls</td>
<td>Picat-sat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance</th>
<th>score</th>
<th>time</th>
<th>score</th>
<th>time</th>
<th>score</th>
<th>time</th>
<th>score</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>warm_up</td>
<td>562.5</td>
<td>0.457</td>
<td>562.5</td>
<td>0.424</td>
<td>562.5</td>
<td>0.892</td>
<td>562.5</td>
<td>t/o</td>
</tr>
<tr>
<td>warm_up*</td>
<td>562.5</td>
<td>0.286</td>
<td>562.5</td>
<td>0.183</td>
<td>562.5</td>
<td>0.615</td>
<td>562.5</td>
<td>t/o</td>
</tr>
<tr>
<td>me_at_the_zoo</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
</tr>
<tr>
<td>trending_today</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
</tr>
<tr>
<td>video_worth</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
</tr>
<tr>
<td>kittens*</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
<td>607.33</td>
<td>56.217</td>
</tr>
</tbody>
</table>

Table 3.1: Results for our Streaming Videos model. (*): MiniZinc 2.1.7, (ψ): MiniZinc 2.2.1.

![Finite Automaton Diagram](image_url)

Figure 3.1: A finite automaton.

Machine assisted reformulations in the future. We produce a specification input language for describing automata with counters. We investigate the tools and languages that are required to produce such compiler. We produce a simple working prototype implementation. The output of the compiler is working decompositions that can directly be used in MiniZinc.

### 3.3.1 Describing an Automaton in MiniZinc

We discuss how to describe an automaton in MiniZinc using regular constraint predicate together with example that is used to illustrate how to describe an automaton in MiniZinc. In [108], a sequence of variables are enforced to take a value which is defined by a finite automaton in regular constraint predicate [116]. The regular predicate has the form \texttt{regular(x, Q, S, d, q0, F)}. The sequence of variables \(x\) is constrained by the deterministic finite automaton (DFA) [86] where \(Q\) is the finite set of states, \(d\) is transition function, initial state \(q0\), and \(F\) is accepting states.
Table 3.2: The encoded transitions of the automaton without counter.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>q1</td>
<td>q2</td>
</tr>
<tr>
<td>q2</td>
<td>q1</td>
<td>q2</td>
</tr>
</tbody>
</table>

Example: Figure 3.1 defines a finite automaton, we define an automaton with two states q1, q2, two symbols 0, 1 on the transitions, the initial state q0 is indicated by an arrow coming from nowhere. A double circle represents an accepting state. The annotated arrows represent the transition function. In the finite automaton, one outgoing arrow with symbol represents each states.

In MiniZinc, the DFA is represented as `regular(x, 2, 2, d, q1, 2)` where Q = 2 states, S = 2 transitions, initial state q0 = q1, accepting states F = 2, and transition function d in Table 3.3.1.

From the given description above, here is the corresponding implementation in MiniZinc.

```mzn
declare int: Q = 2; % states
int: S = 2; % inputs
array[1..Q,1..S] of int: d = % transitions
int: q0 = 1; % initial state
set of int: F = 1..Q; % accepting states
constraint regular(x, Q, S, d, q0, F);
solve satisfy;
output [ "\(x)" ];
```

### 3.3.2 Describing an Automata with Counters

We define an automaton with counters, present an example to specify the issue in describing an counter automaton in MiniZinc. From the example in Section 3.3.1 describing an automaton without counters is straightforward.

Example: Figure 3.2 illustrates another DFA which has states, transitions, and initial state similar to the automaton in Figure 3.1. The main difference is that transitions of the automaton represents the information of
a counter. Transitions without additional information of the counter mean that the counter remains the same value. In this minimal example, we define one counter to specify the issue of representing a automaton with counter in MiniZinc.

In Table 3.3, we present transitions of the automaton in Figure 3.2, each transition $t$ has the form $id_1$, label, $id_2$, operators. The $id_1$, and $id_2$ correspond to the state just before and just after $t$. The label depicts the accepting states to change from state $id_1$ to the next state $id_2$ in order to trigger the transition $t$. The operators represents the corresponding operator that could be applied to the current value of the counter. In this example, we define two usual arithmetic operators that could increment or unchange in the transition $t$, inc or same, respectively.

**Issues** The automaton without counters in Figure 3.1 could be encoded in transition table form, which is supported by MiniZinc. However, it is not any straightforward way to encode the automaton with counters in Figure 3.2 in MiniZinc. We address two issues on specifying an automata with counters in MiniZinc.

Firstly, MiniZinc does not support encoding transition of DFA with counters. At the moment, to the best of our knowledge, there is not any straightforward way to define an automaton with counter in MiniZinc. The current constraint predicate regular consumes the transitions that are encoded in the form of label_1,...,label_s, which corresponds to the input 1..S in Table 3.3.

Secondly, the workaround on specifying an automata manually in Mini-
3.3. Automata with Counters in *MiniZinc*

**Specification of an Automaton with Counters**

**MiniZinc constraint predicate**

**Compiler**

![Diagram](image-url)

Figure 3.3: Reformulation process supports automaton with counters in *MiniZinc*.

*Zinc* is tedious and error-prone. The counter automaton can be defined manually by model constraints for all every single transitions and states. For example, the first transition $t$ in Table 3.3.2 could be defined as constraints in *MiniZinc* by at least 12 lines of code as follows

```plaintext
1   endif
2   % gen_s[1] = 2
3   else
4      if (spec_a[1] = 0)
5         then gen_s[2] = 1 \ gen_c[1] = gen_c[0]
6         else gen_s[2] = 2 \ gen_c[1] = gen_c[0]
7      endif
8   endif
9   \ % gen_s[2] = 1
10  if (gen_s[2] = 1) then
11      if (spec_a[2] = 0)
```

In order to encode all 4 transitions in Table 3.3.2 we need at least 48 lines of code (LOC) in *MiniZinc* in total. It is error-prone since the number of LOC increases corresponding to the number of states and transitions in the automaton.

### 3.3.3 Reformulation Process

We propose a machine-assisted reformulation process to specify the automaton with counters in *MiniZinc*. The reformulation process includes three main components such as the input specification of automaton with counters, the compiler, and the output *MiniZinc* constraint predicate. The reformulation process is illustrated in Figure 3.3.

The reformulation process is in *Type 2: A model with non-CP-standard types is compiled to CP-standard types*, which is showed in Figure 3.4.
Input Specification  The input specification should include all necessary data that could be compiled to MiniZinc model with counter automata. We propose a common signature that contains sufficient data for such compilation process. The arguments of the input specification is described as follows.

- **spec\_a**: input sequence of states, which is an array of integer. E.g. `spec_a = "[0,0,0,1]".`
- **spec\_initState**: initial state.
- **spec\_transitions**: description of transitions and counter’s operations. Each transition has the form \((t, id_i, \text{label}, id_{i+1}, \text{operator})\) where \(t\) is the transition. The \(id_i\) and \(id_{i+1}\) correspond to the state just before and just after \(t\). The \text{label} depicts the accepting states to change from state \(id_i\) to the next state \(id_{i+1}\) in order to trigger the transition \(t\). The\text{operators} represents the corresponding operator that could be applied to the current value of the counter. In this example, we define two usual arithmetic operators that could increment or unchange in the transition \(t\), \text{inc} or \text{same}, respectively.
  - **inc**: increase the counter once, which is equivalent to \(< c_i := c_{i+1} + 1 >\), or \(< c := c + 1 >\) in short.
  - **same**: unchange the counter, which is equivalent to \(< c_i := c_{i+1} >\), or \(< c := c >\) in short. The unchanged counter will be implicit, and not be shown in an automaton.
- **outFileName**: the file name of reformulated result in MiniZinc.

Example: Recall the automaton in Figure3.2. The specification is as follows.

```plaintext
1 spec_a = "[0,1,0,1]"
2```

Figure 3.4: Machine-Assisted Model Reformulation in MiniZinc.
Compiler. The compiler will parse the specification and generate corresponding MiniZinc predicate, which includes common declarations, input data, a predicate, an objective function, output. The compiler is implemented in Haskell. At this stage, the parser will be skipped, the main focus is to compile the given input to the expected output in MiniZinc.

The algorithm to generate the predicate constraint with counter is illustrated in Algorithm 1. To each state \( \text{spec}_a[i] \), a corresponding if-then-else pattern is generated. The generated predicate has the form as follows.

\[
\text{fname(array[int] of int: spec}_a, \text{var int: gen_counter})
\]

Output Constraint Predicate in MiniZinc. The output constraint predicate of example in Figure 3.2, and the Algorithm 1 is described as follows.

```plaintext
var int: gen_counter;
array[int] of int: spec_a;

spec_a = [0,0,0,1];

predicate group_counter(array[int] of int: spec_a, var int: gen_counter) =
  let {
    int: gen_n = length(spec_a),
    array[0..gen_n] of var int: gen_c,
    array[1..gen_n] of var int: gen_s
  in
    gen_c[0] = 0 /
    gen_s[1] = 1
      /
    if (gen_s[1] = 1) then
      if (spec_a[1] = 0)
        then gen_s[2] = 1 /
          gen_c[1] = gen_c[0]
      else gen_s[2] = 2 /
          gen_c[1] = gen_c[0] + 1
    else gen_s[2] = 3 /
          gen_c[1] = gen_c[0] + 1
  end;
```

3.3. Automata with Counters in MiniZinc
Algorithm 1: Algorithm to generate predicate constraint with counter.
3.4 Machine-Assisted Reformulation for MiniZinc

Model reformulation plays an important role in improving models and reducing search space so that solutions can be found faster. In solving CSPs, a model of a CSP may be solved rapidly, while a different model may take excessively long to solve. The efficient solution of CSP is significant in
real-world applications, such as air traffic management, resource allocation, production scheduling, and bioinformatics. Many technologies such as CP, hybrid technologies, MIP, CBLS, SAT could have different solvers and backends to solve the real-time problems. Model reformulation can have a significant impact on solving time. Techniques from formal methods will be used to provide machine assistance for MiniZinc, which is the high-level modelling language to model CSPs. The verification tool, Isabelle, will be used to verify the correctness of reformulations.

We plan to apply recent results in formal methods such as program analysis and synthesis to provide semi-automated frameworks for model analysis. In this thesis, we identify the challenges, implement frameworks, and evaluate our experimental results in reformulations for future research.

Model reformulation [139] plays an important role in improving models and reducing search space so that solutions can be found faster. In solving CSPs [62], a model of a CSP may be solved rapidly, while a different model may take excessively long to solve. The efficient solution of CSP is significant in real-world applications, such as air traffic management, resource allocation [103], production scheduling, and bioinformatics [11]. Because such problems are often NP-hard, all known algorithms have exponential-time worst-case behaviour. Many technologies such as CP, MIP [85], CBLS, SMT [14][110], and SAT [14] could have different solvers and backends to solve the real-time problems. Each technology has its scope of application, and none of the technology is dominant all problems. CP [128] is pervasive and widely used to solve real-time problems which input data could be scaled up to the enormous sizes, and results are required to be given efficiently and dynamically.

CSPs can be reformulated automatically or semi-automatically. The automatic reformulation of CSPs can be obtained using various methodologies and approaches [20][27][30][100][101][102]. The automation in modelling has been advanced in five areas such as (1) generating a kernel, in which the compilation in a direct or non-direct system, (2) identifying symmetries, (3) breaking symmetries, (4) adding implied constraints, and (5) transforming constraints. These five advances are discussed in an invited talk at the 10th international workshop on constraint modelling and reformulation [63]. In addition, the reformulation may be achieved semi-automatically [4][35][36]. One approach is using a learning-based reformulation approach [164] or improving and stepping towards the automatic methods work [136].

The reformulation of a model into a logically equivalent model is a valuable tool to assist a modeller. A correct reformulation can significantly improve the solving time with the confidence of modeller. The Global Constraint Catalogue [16], the global-constraint library of the MiniZinc system [108], and the Essence system [64][111] are a few resources for refor-
mulation. Both MiniZinc [108] and Essence [64, 111] are solver-independent languages for modelling CSPs with many common global constraints. In addition, their reformulations are used when a target solver does not have the required globals. Reformulations are inferred in both the Model Seeker [18] and the Globalizer [96] by testing many combinations of global constraints on possible input data and then ranking the possible reformulations.

No proofs of correctness are provided for these reformulations, and the modeller must decide the correctness of the suggested reformulations. No work has been done on verifying the correctness of such reformulation rules, although there are tools [64, 108, 111] that could generate reformulation rules.

Interactive theorem provers are notoriously difficult to use. Moreover, the development of proof tactics can be difficult in a new application area such as CP. However, even suggesting and verifying only a few reformulation rules will remarkably benefit modellers of CP and will still be a major advance over state of the art. Machine assistance that provides verified reformulation rules allows modellers to improve their models by reformulations with confidence. The future contributions are as follows

- Develop general-purpose tactics for reformulations of models in CP.
- Propose proof systems to prove soundness and completeness of reformulation rules.
- Provide semi-automated and automated frameworks for model analysis, synthesis, and refinement.
- Implement a library of automatic generation of verified reformulation rules for the MiniZinc toolchain.

The key objectives of the Ph.D. project are to support modellers to generate equivalent models that are verified and solve CSPs faster. The novelty and originality of the project are that although many constraint modelling toolchains support to generate reformulation rules [64, 108, 111], there is not any work that has been done on verifying the correctness of such generated reformulation rules. For instance, with possible input date, the Model Seeker [18] and the Globalizer [96] test many combinations of global constraints. Then, all possible reformulations are ranked and finally provided to modellers without verifying their soundness and completeness. Applying theorem proving techniques and verifying reformulation rules crucially differentiate our work from other existing works. Even though just a few generated reformulation rules is verified, it could be a significant advance to non-expert modellers over state of the art.

When writing this thesis, it does not exist any model reformulation system with the auto-generated proof of models equivalence. The problem con-
sists of several components and research questions that are needed to solve gradually and incrementally. It requires comprehensive knowledge, which is related to programming languages, theorem proving techniques, optimisation methods, and machine learning. In order to solve the problem, we divide the problems into sub-problems with corresponding questions. For instances, (1) how to find the implied constraints in a given model? (2) how to assert that the improved model with supplemental implied constraints is better than the original model? (3) how to automatically derive the formulas of basic and improved models? (4) what and how the translation between modelling language and proof language? (5) how to develop an integrated system that takes the original model, produces an improved model with the proof of equivalence between these models?

To the engineering aspects, the system should contain four components as follows:

- Preprocessing component: to transform the model to corresponding theorem prover input and vice versa.
- Reformulation component: to produce an improved model from a given model.
- Theorem prover component: to prove the soundness and completeness of two models.
- Intermediate language component: to translate between modelling language and theorem proving representation.
Chapter 4

Conclusions and Future Work

Contributions and Limitations We present a working prototype compiler that takes the given input of DFA, and produces a corresponding constraint predicate in MiniZinc. This is the foundation to extend our compiler to deal with more states and transitions of any given DFA.

There are some restrictions in the number of states and transitions in the input DFA specification. The compiler takes the input specification directly and skips the parsing phase.

Conclusions and Future Work In the scope of the project, we aim to answer all questions as mentioned above. In the initial phase, we firstly focus on solving the problem manually. In the second phase, we selectively improve some manual components to semi-automatic components. In the third phase, we tackle the automated-components problem. Finally, we accomplish the project by integrating and benchmarking the system.

In the first phase, MiniZinc’s reformulation rules will be formalised in Isabelle [115] or a similar system. We take advantage of Isabelle theorem prover to prove the soundness and completeness of two models. We aim to develop general-purpose tactics for the reformulation of models. We investigate more specialised theories that will be useful in proving the correctness and soundness in the second phase. We will investigate logics for proving the correctness and soundness of reformulation rules and their integration into MiniZinc. Finally, we will benchmark the system using several case studies that could be found from MiniZinc competition [144], and CSPLib [138].
Bibliography


[111] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in Savile...


Bibliography


Recent licentiate theses from the Department of Information Technology

2019-006  Kristiina Ausmees: Efficient Computational Methods for Applications in Genomics
2019-005  Carl Jidling: Tailoring Gaussian Processes for Tomographic Reconstruction
2019-004  Amin Kaveh: Local Measures for Probabilistic Networks
2019-001  Diane Golay: An Invisible Burden: An Experience-Based Approach to Nurses’ Daily Work Life with Healthcare Information Technology
2018-004  Charalampos Orfanidis: Robustness in Low Power Wide Area Networks
2018-003  Fredrik Olsson: Modeling and Assessment of Human Balance and Movement Disorders Using Inertial Sensors
2018-002  Tatiana Chistiakova: Ammonium Based Aeration Control in Wastewater Treatment Plants - Modelling and Controller Design
2018-001  Kim-Anh Tran: Static Instruction Scheduling for High Performance on Energy-Efficient Processors