Reformulations of Constraint Satisfaction Problems: A Survey

Huu-Phuc Vo[1][0000−0003−2106−7186]

Dept. of Information Technology, Uppsala University
Uppsala, Sweden
huu-phuc.vo@it.uu.se

Table of Contents

List of Figures

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 survey, 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.

Keywords: reformulation, constraint programming, transformation

1 Introduction

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 [8,78,127,140]. 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 [123], Choco [120], GECODE solver [74], HAMPI [93, 94], KALUZA [130], SUSHI [73], Z3 [47]. There are logic programming based constraint modelling languages and toolchain such as ECLiPSe [12], SICStus Prolog [33], ESSENCE [67, 68], and CONJURE [5]. Model reformulation plays an important role in improving models, reducing search space so that solutions can be found faster [114]. Hence we categorise model reformulation into three types: that a model is reformulated to another model with the same modelling language [140, 148], that a model with non CP standard types is compiled to another model with CP standard types [79], 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) [79, 112, 138]. 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 survey, 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 survey contrasts and compares different categories of reformulation for solving CSPs as well as ways of applying these categories.

Reformulation types	**References**
Automatic reformulations	[6, 16, 18, 23, 31, 54, 69, 82, 103] [97, 104, 105, 113, 143, 157]
Semi-automatic reformulation	[4, 38, 39, 163]
Reformulation by changing viewpoint	[7, 88, 107, 141]
Reformulate MIP to CP	[22, 41, 46, 92, 96, 149, 151, 159]
Reformulate implied constraints	[63, 64, 86]
Reformulate Set CSPs to CSPs	[3, 14, 44, 45, 81, 109]
Reformulate String variables to CSP without String variables	[2, 9, 73, 80, 89, 94, 132, 133, 134]
Reformulate by symmetry breaking	[127]
Reformulate by pre-computation	[1, 35, 49, 50, 77, 90, 98, 108]
Reformulate non-binary to binary translations	[91, 161]
Reformulation into SAT	[26, 54, 76, 115, 153]
Reformulation into MILP	[79, 112, 138]
Reformulation into SMT	[10, 138]
Other reformulation approaches	[19, 26, 70, 87, 99, 102, 135, 137, 146] [13, 36, 71, 115, 124, 142, 156, 162] [30, 56, 87, 154]

Table 1. References of reformulation types

2 Preliminaries

In this section, we describe the preliminary concepts such as CP, CSPs, reformulations of CSPs, and modelling languages. The section is partly based on [78, 79, 138, 140, 147].

2.1 Constraint Programming

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 [65]. 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 [127], the authors discuss the historical foundations and an overview of all aspects of CP. A more pedagogic approach to CP appears in [11, 48, 101].

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 [65]. 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

Example: Nurse Rostering Problem [119, 128] Consider the example of nurse rostering problem [106]. 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.

Table 2. Nurse Rostering Problem.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse A	call	none	oper	none	oper	none	none
Nurse B	app	call	none	oper	none	none	call
Nurse C	oper	none	call	app	app	call	none
Nurse D	app	oper	none	call	oper	none	none
Nurse E	oper	none	oper	none	call	none	none

In the nurse rostering problem, we model the problem using the *MiniZinc*-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 [110].

```
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));
14 % ... other constraints
```

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 1 illustrates the whole process.

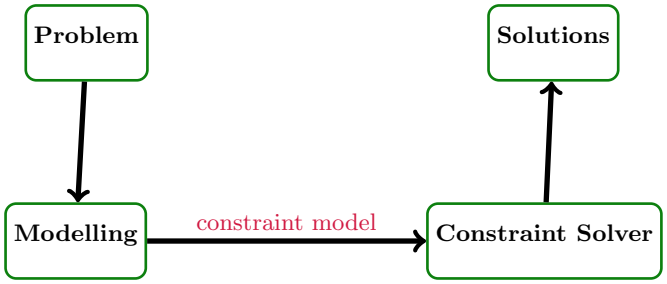

Fig. 1. A process of finding solutions from given natural statements of problems.

Figure 2 contrasts the paradigms of modelling and programming approaches [117]. 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 3.

In the pipelining process, the models may be compiled to multiple solvers. Figure 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, 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 5 shows the pipelining process to multi-solvers of ESSENCE.

Example of MiniZinc Pipelining process Figure 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.

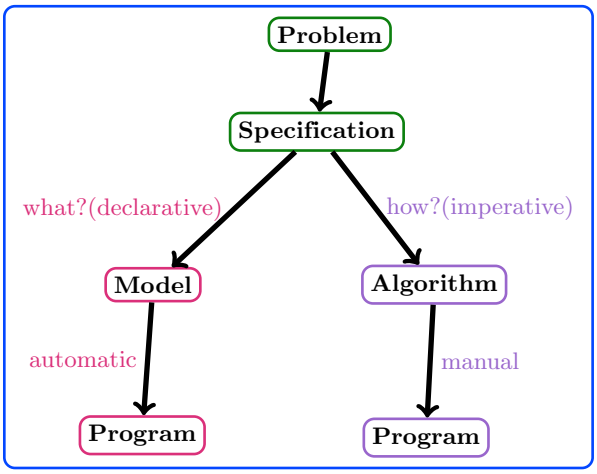

Fig. 2. Modelling and programming approaches.

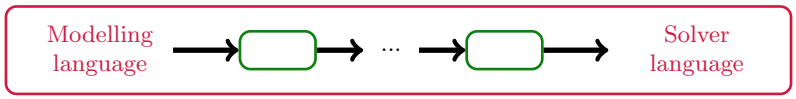

Fig. 3. General Pipelining Process to one-solver.

2.4 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 [84]. 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 [140, 148]. The *second type* is that a model with non-CP standard types is compiled to another model with CP standard types [79]. The *third type* is that a model is converted to SAT, MIP, SMT, or others [54, 79, 112, 138].

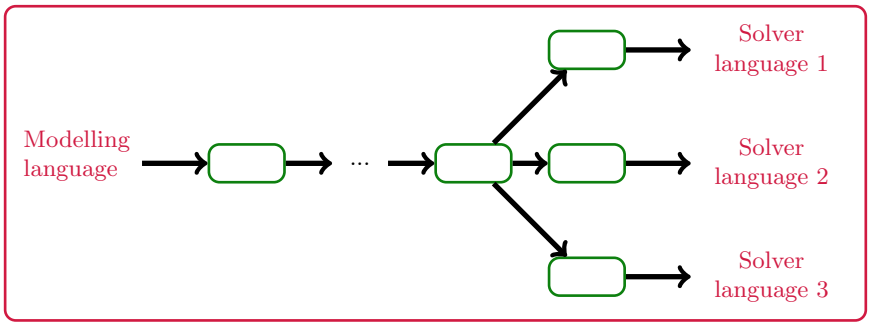

Fig. 4. General Pipelining Process to multi-solvers.

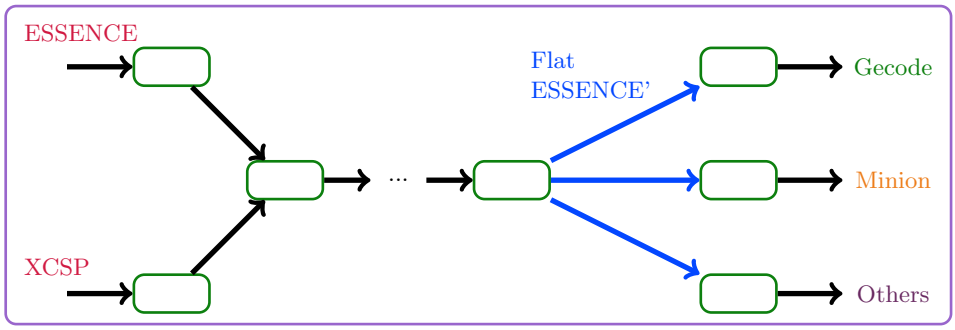

Fig. 5. ESSENCE Pipelining Process to multi-solvers.

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 [140], 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 senquencing problem* [139]: 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.

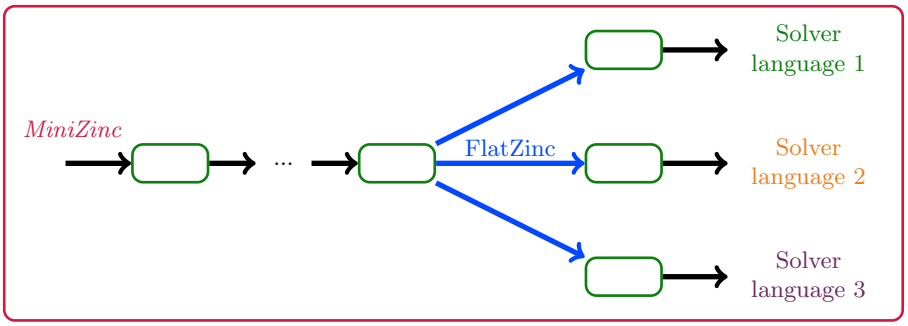

Fig. 6. *MiniZinc*Pipelining Process to multi-solvers.

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 [75]. 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 reduces the set of solutions by elimiating 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.

– Generic Reformulations (Section 3.1).
  - **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

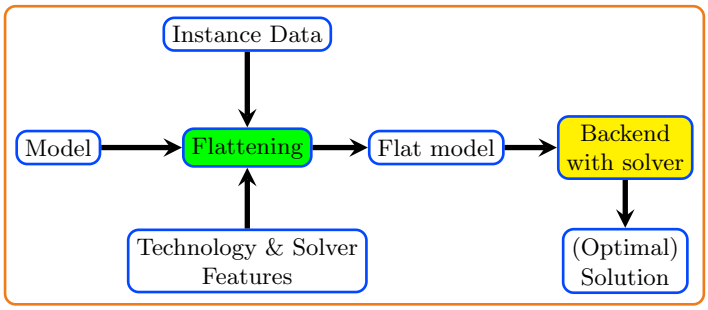

Fig. 7. MiniZinc toolchain.

notoriously difficult to formulate [160], 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 [41]. MIP is applicable for many classes of combinatorial problems, which can be subdivided into sequencing problems and allocation problems [22, 46, 92, 96, 149, 151, 158, 159]

2.5 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 [111] 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 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 [74], ECLiPSe [12], SICStus Prolog [33], and a number of solvers, e.g., Zinc [17, 100] and Mercury [20], developed by the G12 research team [144]. The backend with solvers takes the FlatZinc model to give the (optimal) solution. *MiniZinc* toolchain is illustrated in Figure 7 [118].

Example: Job-shop problem [51, 62, 164, 165] 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 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 [138].

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

```
size = 2; d = [| 2,5 | 3,4 |];
```

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

```
int: size; % size of problem, number of jobs and machines
array [1..size,1..size] of int: d; % task durations
int: total = sum(i,j in 1..size) (d[i,j]); % total
  duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end; % total end time
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 [111]. In the flattening part, several reductions such as built-ins evaluation, comprehension unrolling, compound built-in unrolling, fixed array 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 assignment 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 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 survey on the reformulation, we could name

some of modelling languages in the past such as *OPL* [150], *ESRA* [59], *NP-Spec* [29], *F and Fiona* [85], *CGRASS* [72], *TAILOR and SAVILE ROW* [125], [17, 100], [145], and *CONJURE 1.0* [4].

Other modelling languages There are more modelling languages other than *MiniZinc* such as ESSENCE and ESSENCE' [67], AIMMS [24], GAMS [25], AMPL [61], Mosel-Xpress [40], SMT-lib [27], 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 features 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.

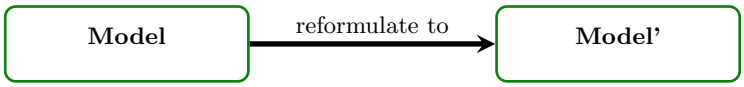

Fig. 8. A model is reformulated to another model' that is better than the original model.

3 Reformulations

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 2.4, 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.

3.1 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 [140, 148], which is illustrated in Figure 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 [124].

Type 2 A model with non-CP standard types is compiled to another model with CP standard types, which is demonstrated in Figure 9. In Chapter 17 of the Handbook of CP [79], 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 [9], the authors introduce the extension of the

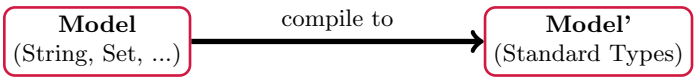

Fig. 9. A model with non-CP-standard types is compiled to CP-standard types.

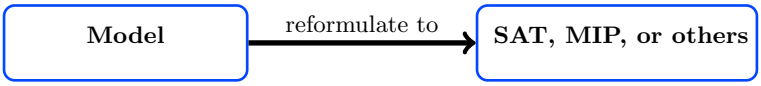

Fig. 10. A model is reformulated to SAT, MIP, or others.

MiniZinc that can specify string variables of unknown length. A new extension, called Gecode+S [131], which is an official part of Gecode, with bounded-length string variables of Gecode solver [74] also supports string constraints [44, 45].

Type 3 A model is reformulated into SAT, MIP, MILP, or SMT, which is represented in Figure 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) [58]. Moreover, two encodings of binary CSPs are examined into SAT in [54]. 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 [79]. One can model the CSPs as a propositional satisfiability formula, which can be solved by SAT solvers [112]. In [138], the authors develop two systems to reformulate CSPs into SMT instances, which can be solved using SMT solvers. The reformulation into SAT, MILP, and SMT formalisms is briefly introduced in [138]. CSPs can be solved in a different way by reformulating them into other common formalisms such as SAT, MILP, and SMT.

Tailor [125] 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.

3.2 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 [54], 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 [16], the authors describe a prototype modelling system that automatically refines an abstract specification of a CSP into a set of alternative constraint programs [23, 31, 103]. By exploiting dominance relations in constraint optimisation problems to dramatically reduce the search space, [104,105] propose an automatic method to detect some of the dominance relations manually identified in [37]. In [97], 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 [143], 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 [157], which generates effective streamliners automatically from the specification of ESSENCE [68, 69]. 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 [66]. 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 [38,39]. 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 [163]. 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.

3.3 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 [7, 88, 107, 141]. In [36], 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 [102], 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 [34].

Example: Objects, Shapes, and Colours problem [32, 95, 118] 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 [118] illustrate three different viewpoints.

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).

```
1 % Viewpoint 1: Which colour, if any, does each shape have?
2
3 int: n; % number of objects
4 int: s; % number of shapes
5 int: c; % number of colours
6 constraint assert(s >= n, "Not enough shapes");
7
8 array[1..s] of var 0..c: Colour; % 0 is a dummy colour
9 % There are n objects:
10 constraint count(Colour,0) = s-n;
11 % The numbers of objects of the used colours are distinct:
12 constraint
    alldifferent_except_0(global_cardinality(Colour,1..c));
13 % The objects have distinct shapes:
14 % implied by lines 6 and 9
15 % ... add here the other constraints ...
16 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).

```
1 % Viewpoint 2: Which shapes, if any, does each colour
    have?
2
3 int: n; % number of objects
```

[1] http://www.csplib.org/

```
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;
% There are n objects:
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.

```
% 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;
```

3.4 Reformulate Implied Constraints

Implied constraints or redundant constraints are constraints, which are implied by the constraints defining the problem [140]. 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, [52] 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 [140]. In [64], the

authors construct implied constraints for automaton constraints based on their earlier work [63, 86], 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 [21] of a constraint on a sequence of variables.

Example: Magic Series problem [152] To illustrate the reformulation using implied constraints, we consider the *Magic Series* problem as an example. 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 $Magic = [1, 2, 1, 0]$. The domain of each variable of the $Magic$ array is $[0...n]$. Here is the constraint which is directly from [118].

```
1 global_cardinality_closed(Magic,I,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.

```
1 sum(Magic)=n /\ sum(i in I)(Magic[i]*i)=n
```

3.5 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 [78]. Symmetry breaking plays an important role in speeding up the search in CSPs that contain symmetry [28,43,57,60,116,122,126]. 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 [121]. Another simple method for breaking any type of symmetry between variables has been presented in [42]. To deal with symmetries, in [155], 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 [127]. In extreme cases, reformulation of problems can be critical in dealing with symmetries.

Example: Social Golfers Problem [53, 83] 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 $Weeks(w) \times Groups(g) \times Slots(s) \rightarrow Playters(g \cdot s)$ 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 3.

	Group 1	Group 2	Group 3	Group 4
Week 1	[1,2,3]	[4,5,6]	[7,8,9]	[10,11,12]
Week 2	[1,4,7]	[2,5,10]	[3,8,11]	[6,9,12]
Week 3	[1,8,10]	[2,4,12]	[3,5,9]	[6,7,11]
Week 4	[1,9,11]	[2,6,8]	[3,4,10]	[5,7,12]

Table 3. A solution for the Social Golfers Problem for $<w, g, s> = <4, 4, 3>$.

By observing Table 3, 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 $3!^{16}$ 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 3 to a 2D $w \times g$ array of set variables as in Table 4 and adding the constraint that all sets must be of size s.

	Group 1	Group 2	Group 3	Group 4
Week 1	{1,2,3}	{4,5,6}	{7,8,9}	{10,11,12}
Week 2	{1,4,7}	{2,5,10}	{3,8,11}	{6,9,12}
Week 3	{1,8,10}	{2,4,12}	{3,5,9}	{6,7,11}
Week 4	{1,9,11}	{2,6,8}	{3,4,10}	{5,7,12}

Table 4. A symmetry breaking solution for the Social Golfers Problem for $<w, g, s> = <4, 4, 3>$.

3.6 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 [49]. Their study is extended in another publication in 2017 using auto-tabling, which replaces model parts by constraints with precomputed solution arrays [50]. 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 [98] 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 [90]. 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 [77], multivalued decision diagrams [35], and extended indexicals [108].

Example: Prize-pool Division problem [118] 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.

```
1  ...
2 array[1..5] of int: Pools = [1000,5000,15000,20000,25000];
3 var 1..5: x;
4 var 1..500: numWinners;
5 ...
6 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 $numWinnders$, we pre-compute a 2D array, which is indexed by 1..5 and 1..500.

```
1 ...
2 array[1..5] of int: Pools = [1000,5000,15000,20000,25000];
3 var 1..5: x;
4 var 1..500: numWinners;
5 ...
6 array[1..5,1..500] of int: objFun =
7     array2d(1..5,1..500,
8             [Pools[p] div n | p in 1..5, n in 1..500]
9            );
10 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.

3.7 Reformulate SET-CSPs to CSPs

In [44,45], 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, 81], 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 [14], 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 [109, 129], 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.

3.8 Reformulate String Variables

A substantial amount of research in recent years on the development of solvers for string constraints has been done. In [9], 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 [9]. 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 [73], 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 [94]. 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 [89], 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 [80, 132, 133, 134].

3.9 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 [140]. 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 [15] 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 [161], 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 [91], 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 [55], 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.

3.10 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 [54], 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 [153], and the supported encoding [76]. They claim that applying their hyper-resolution reduces both the runtime and the number of search nodes. In [26,115], 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.

```
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 ($/\backslash$) of clauses. A clause is a disjunction ($\backslash/$) 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 $w = \mathit{false}, x = \mathit{true}, y = \mathit{true}$, and $z = \mathit{false}$.

3.11 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. [79,112,138].

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 [138], the authors develop two systems to reformulate CSPs into SMT instances, which can be solved using SMT solvers.

Acknowledgement I would like to thank Justin Pearson, Di Yuan, and Pierre Flener for comments and feedback.

References

1. Abdennadher, S., Rigotti, C.: Automatic generation of rule-based constraint solvers over finite domains. ACM Trans. Comput. Logic **5**(2), 177–205 (Apr 2004). https://doi.org/10.1145/976706.976707, http://doi.acm.org/10.1145/976706.976707
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9
3. Ågren, M.: Set Constraints for Local Search. Ph.D. thesis, Department of Information Technology, Uppsala University, Sweden (2008), available at http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-8373
4. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
5. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 107–116. Springer (2013)
6. Akgun, O., Frisch, A.M., Hnich, B., Jefferson, C., Miguel, I.: Conjure revisited: Towards automated constraint modelling. arXiv preprint arXiv:1109.1774 (2011)
7. Akgün, Ö., Miguel, I.: Modelling langford's problem: A viewpoint for search. arXiv preprint arXiv:1808.09847 (2018)
8. Allignol, C., Barnier, N., Flener, P., Pearson, J.: Constraint programming for air traffic management: A survey. The Knowledge Engineering Review **27**(3), 361–392 (September 2012)
9. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with strings. In: Hermenegildo, M., López-García, P. (eds.) LOPSTR 2016: Revised Selected Papers. pp. 59–75. No. 10184 in LNCS, Springer (2017)
10. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving weighted csps with meta-constraints by reformulation into satisfiability modulo theories. Constraints **18**(2), 236–268 (Apr 2013). https://doi.org/10.1007/s10601-012-9131-1, https://doi.org/10.1007/s10601-012-9131-1
11. Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)
12. Apt, K.R., Wallace, M.: Constraint Logic Programming using ECLiPSe. Cambridge University Press (2006)
13. Armant, V., Brown, K.N.: Reformulation of drivers' fixed path constraints in ridesharing problems (2016)
14. Azevedo, F., Barahona, P.: Applications of an extended set constraint solver. In: ERCIM / CompulogNet Workshop on Constraints (2000)
15. Bacchus, F., Van Beek, P.: On the conversion between non-binary and binary constraint satisfaction problems. In: AAAI/IAAI. pp. 310–318 (1998)
16. Bakewell, A., Frisch, A.M., Miguel, I.: Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In: 2nd International Workshop on Modelling and Reformulating CSPs. pp. 3–17 (2003), http://www-users.cs.york.ac.uk/~frisch/Reformulation/03
17. de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: The modelling language zinc. In: International Conference on Principles and Practice of Constraint Programming. pp. 700–705. Springer (2006)

18. Baumgartner, P., Slaney, J.: Constraint modelling: A challenge for automated reasoning. First-order Theorem Proving FTP 2009 p. 4 (2009)
19. Beck, J.C., Prosser, P., Selensky, E.: An empirical study of mutual routing-scheduling reformulation. Modelling and Reformulating Constraint Satisfaction Problems p. 18 (2003)
20. Becket, R., Brand, S., Brown, M., Duck, G.J., Feydy, T., Fischer, J., Huang, J., Marriott, K., Nethercote, N., Puchinger, J., et al.: The many roads leading to rome: Solving zinc models by various solvers. In: 7th International Workshop on Constraint Modelling and Reformulation (2008)
21. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer (2004)
22. Bertsimas, D., Lulli, G., Odoni, A.: The air traffic flow management problem: An integer optimization approach. Integer Programming and Combinatorial Optimization **5035**(4), 34–46 (May 2008)
23. Bessiere, C., Quinqueton, J., Raymond, G.: Mining historical data to build constraint viewpoints. In: 5th International Workshop on Constraint Modelling and Reformulation. pp. 1–16 (2006)
24. Bisschop, J.: AIMMS Optimization Modeling. Lulu.com (2012), available at `http://www.aimms.com/downloads/manuals/optimization-modeling`
25. Bisschop, J., Meeraus, A.: On the development of a general algebraic modeling system in a strategic planning environment. Mathematical Programming Study pp. 1–29 (1982), see `http://gams.com`
26. Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving intensional weighted csps by incremental optimization with bdds. In: International Conference on Principles and Practice of Constraint Programming. pp. 207–223. Springer (2014)
27. Bofill, M., Suy, J., Villaret, M.: A system for solving constraint satisfaction problems with SMT. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 300–305. Springer (2010)
28. Brown, C.A., Finkelstein, L., Purdom Jr, P.W.: Backtrack searching in the presence of symmetry. In: Mora, T. (ed.) 6th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. LNCS, vol. 357, pp. 99–110. Springer (1988)
29. Cadoli, M., Ianni, G., Palopoli, L., Schaerf, A., Vasile, D.: Np-spec: An executable specification language for solving all problems in np. Computer Languages **26**(2-4), 165–195 (2000)
30. Cadoli, M., Mancini, T.: Exploiting functional dependencies in declarative problem specifications. In: European Workshop on Logics in Artificial Intelligence. pp. 628–640. Springer (2004)
31. Cadoli, M., Mancini, T.: Automated reformulation of specifications by safe delay of constraints. Artificial Intelligence **170**(8), 779–801 (2006)
32. Carlsson, M., Beldiceanu, N., Martin, J.: A geometric constraint over k-dimensional objects and shapes subject to business rules. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 220–234. Springer (2008)
33. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer (1997)
34. Cheng, B., Choi, K., Lee, J., Wu, J.: Increasing constraint propagation by redundant modeling: An experience report. Constraints **4**, 167–192 (1999)

35. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 509–523. Springer (2008)
36. Choi, C.W., Lee, J.H.M., Stuckey, P.J.: Propagation redundancy in redundant modelling. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming – CP 2003. pp. 229–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
37. Chu, G., Stuckey, P.J.: A generic method for identifying and exploiting dominance relations. In: Principles and Practice of Constraint Programming. pp. 6–22. Springer (2012)
38. Coletta, R., Bessiere, C., O'Sullivan, B., Freuder, E.C., O'Connell, S., Quinqueton, J.: Semi-automatic modeling by constraint acquisition. In: International Conference on Principles and Practice of Constraint Programming. pp. 812–816. Springer (2003)
39. Coletta, R., Bessiere, C., O'Sullivan, B., Freuder, E.C., O'Connell, S., Quinqueton, J.: Constraint acquisition as semi-automatic modeling. In: Research and Development in Intelligent Systems XX, pp. 111–124. Springer (2004)
40. Colombani, Y., Heipcke, S.: Mosel: An extensible environment for modeling and programming solutions. In: Jussien, N., Laburthe, F. (eds.) CP-AI-OR 2002 (2002), http://www.emn.fr/z-info/cpaior
41. Côté, M.C., Gendron, B., Rousseau, L.M.: Modeling the Regular constraint with integer programming. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CP-AI-OR 2007. LNCS, vol. 4510, pp. 29–43. Springer (2007)
42. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. KR **96**, 148–159 (1996)
43. Crawford, J.M., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Carlucci Aiello, L., Doyle, J., Shapiro, S.C. (eds.) KR 1996. pp. 148–159. Morgan Kaufmann (1996)
44. Cristiá, M., Rossi, G.: A decision procedure for sets, binary relations and partial functions. In: International Conference on Computer Aided Verification. pp. 179–198. Springer (2016)
45. Cristiá, M., Rossi, G., Frydman, C.: Adding partial functions to constraint logic programming with sets. Theory and Practice of Logic Programming **15**(4-5), 651–665 (2015)
46. Darby-Dowman, K., Little, J.: Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. INFORMS Journal on Computing **10**(3), 276–286 (1998)
47. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
48. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
49. Dekker, J.J.: Sub-Problem Pre-Solving in MiniZinc. Master's thesis, Department of Information Technology, Uppsala University, Sweden (November 2016), available at http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-307145
50. Dekker, J.J., Björdal, G., Carlsson, M., Flener, P., Monette, J.N.: Auto-tabling for subproblem presolving in MiniZinc. Constraints **22**(4), 512–529 (October 2017)
51. Dell'Amico, M., Trubian, M.: Applying tabu search to the job-shop scheduling problem. Annals of Operations Research **41**(1-4), 231–252 (1993)
52. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Kodratoff, Y. (ed.) ECAI 1988. pp. 290–295. Pitman (1988)

53. Dotú, I., Van Hentenryck, P.: Scheduling social tournaments locally. AI Communications **20**(3), 151–162 (2007)
54. Drake, L., Frisch, A.M., Gent, I., Walsh, T.: Automatically reformulating sat-encoded csps (2002)
55. Dubois, C.: Formally verified decomposition of non-binary constraints into equivalent binary constraints. Journées Francophones des Langages Applicatifs 2019 p. 237
56. Flener, P.: Towards relational modelling of combinatorial optimisation problems. In: Bessière, C. (ed.) IJCAI 2001 Workshop on Modelling and Solving Problems with Constraints. pp. 31–38 (2001), available at `http://www.lirmm.fr/~bessiere/ws_ijcai01`
57. Flener, P., Frisch, A.M., Hnich, B., Kızıltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer (2002)
58. Flener, P., Monette, J.N.: Solving technologies (2018), available at `http://user.it.uu.se/~pierref/courses/COCP/lectures.html`
59. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Bruynooghe, M. (ed.) LOPSTR 2003: Revised Selected Papers. LNCS, vol. 3018, pp. 214–232. Springer (2004)
60. Flener, P., Pearson, J., Sellmann, M., Van Hentenryck, P., Ågren, M.: Structural symmetry breaking for constraint satisfaction problems. In: Gent, I.P., Linton, S. (eds.) International Symmetry Conference, Edinburgh, UK (2007)
61. Fourer, R., Gay, D.M., W., K.B.: AMPL: A Modeling Language for Mathematical Programming. Cengage Learning, second edn. (2002), available at `http://ampl.com/resources/the-ampl-book`
62. Fox, M.S.: Constraint-Directed Search: A Case Study of Job-Shop Scheduling. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, USA (December 1983)
63. Francisco Rodríguez, M.A., Flener, P., Pearson, J.: Generation of implied constraints for automaton-induced decompositions. In: Brodsky, A. (ed.) ICTAI 2013. pp. 1076–1083. IEEE Computer Society (2013)
64. Francisco Rodríguez, M.A., Flener, P., Pearson, J.: Implied constraints for Automaton constraints. In: Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.) GCAI 2015: Global Conference on Artificial Intelligence. EasyChair Proceedings in Computing, vol. 36, pp. 113–126 (2015)
65. Freuder, E.C., Mackworth, A.K.: Constraint satisfaction: An emerging paradigm. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 2, pp. 11–26. Elsevier (2006)
66. Frisch, A.M.: A decade of progress in constraint modelling and reformulation the quest for abstraction and automation (2011), available at `https://www-users.cs.york.ac.uk/~frisch/Research/decade.pdf`
67. Frisch, A.M., Grum, M., Jefferson, C., Martinez Hernandez, B., Miguel, I.: The design of Essence: A constraint language for specifying combinatorial problems. In: IJCAI 2007. pp. 80–87. Morgan Kaufmann (2007)
68. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: E ssence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
69. Frisch, A.M., Hnich, B., Miguel, I., Smith, B.M., Walsh, T.: Towards csp model reformulation at multiple levels of abstraction (2002)
70. Frisch, A.M., Jefferson, C., Hernández, B.M., Miguel, I.: The rules of modelling: Towards automatic generation of constraint programs (2004)

71. Frisch, A.M., Jefferson, C., Hernández, B.M., Miguel, I.: The rules of constraint modelling. In: IJCAI. pp. 109–116 (2005)
72. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: A system for transforming constraint satisfaction problems. In: O'Sullivan, B. (ed.) Recent Advances in Constraint. LNAI, vol. 2627, pp. 15–30. Springer (2003)
73. Fu, X., Powell, M.C., Bantegui, M., Li, C.C.: Simple linear string constraints. Formal Aspects of Computing **25**, 847–891 (November 2013). https://doi.org/10.1007/s00165-011-0214-3, SUSHI is available at http://people.hofstra.edu/Xiang_Fu/XiangFu/projects/SAFELI/SUSHI.php
74. Gecode Team: Gecode: A generic constraint development environment (2018), the Gecode solver and MiniZinc backend are available at http://www.gecode.org
75. Gent, I.P.: Two results on car-sequencing problems. Report University of Strathclyde, APES-02-98 **7** (1998)
76. Gent, I.P.: Arc consistency in sat. In: ECAI. vol. 2, pp. 121–125 (2002)
77. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. Artificial Intelligence **211**, 1–33 (2014)
78. Gent, I.P., Petrie, K.E., Puget, J.F.: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 10, pp. 329–376. Elsevier (2006)
79. Gervet, C.: Constraint over structured domains. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 17, pp. 603–636. Elsevier (2006)
80. Golden, K., Pang, W.: Constraint reasoning over strings. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 377–391. Springer (2003)
81. Gumucio E., R.R.: Constraints on Set Variables for Constraint-based Local Search. Master's thesis, Department of Information Technology, Uppsala University, Sweden (2011), available at http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-159180
82. Guns, T., Dries, A., Tack, G., Nijssen, S., De Raedt, L.: Automatic solver chaining in miningzinc. In: Proceedings of the Fourteenth International Workshop on Constraint Modelling and Reformulation (ModRef). pp. 1–17 (2015)
83. Harvey, W.: CSPLib problem 010: Social golfers problem. http://www.csplib.org/Problems/prob010
84. Hnich, B.: High-level modelling and reformulation of constraint satisfaction problems. Lecture notes in computer science pp. 766–766 (2001)
85. Hnich, B.: Function variables for constraint programming. AI Communications **16**(2), 131–132 (2003)
86. Hnich, B., Richardson, J., Flener, P.: Towards automatic generation and evaluation of implied constraints. Tech. Rep. 2003-014, Department of Information Technology, Uppsala University, Sweden (originally written in August 2000), available at http://www.it.uu.se/research/reports/2003-014
87. Hnich, B., Smith, B.M., Walsh, T.: Dual modelling of permutation and injection problems. Journal of Artificial Intelligence Research **21**, 357–391 (2004)
88. Hnich, B., Walsh, T.: Why channel? multiple viewpoints for branching heuristics. Modelling and Reformulating Constraint Satisfaction Problems p. 94 (2003)
89. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Automated Software Engineering **19**(4), 531–559 (2012)
90. IBM Knowledge Center: The strong constraint, available at http://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.

odms.ide.help/OPL_Studio/opllang_quickref/topics/tlr_oplsch_strong.html
91. Järvisalo, M., Niemelä, I.: A compact reformulation of propositional satisfiability as binary constraint satisfaction. Modelling and Reformulating Constraint Satisfaction Problems pp. 111–124 (2004)
92. Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P.: Progress in linear programming-based algorithms for integer programming: An exposition. INFORMS Journal on Computing **12**(1), 2–23 (2000)
93. Kieżun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology **21**(4), article 25 (2012)
94. Kieżun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Rothermel, G., Dillon, L.K. (eds.) ISSTA 2009. pp. 105–116. ACM (2009), HAMPI is available at http://people.csail.mit.edu/akiezun/hampi/
95. Kubale, M., Jackowski, B.: A generalized implicit enumeration algorithm for graph coloring. Communications of the ACM **28**(4), 412–418 (1985)
96. Law, Y.C., Lee, J.H.M.: Global constraints for integer and set value precedence. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 362–376. Springer (2004)
97. Le Charlier, B., Khong, M.T., Lecoutre, C., Deville, Y.: Automatic synthesis of smart table constraints by abstraction of table constraints. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. pp. 681–687. IJCAI'17, AAAI Press (2017), http://dl.acm.org/citation.cfm?id=3171642.3171740
98. Le Provost, T., Wallace, M.: Domain independent propagation. In: FGCS 1992, International Conference on Fifth Generation Computer Systems. pp. 1004–1011. IOS Press (1992)
99. Little, J., Gebruers, C., Bridge, D., Freuder, E.: Capturing constraint programming experience: A case-based approach. In: CP-02 Workshop on Reformulating Constraint Satisfaction Problems (2002)
100. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., De La Banda, M.G., Wallace, M.: The design of the zinc modelling language. Constraints **13**(3), 229–267 (2008)
101. Marriott, K., Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press (1998)
102. Martınez-Hernández, B., Frisch, A.M.: The systematic generation of channelling constraints. Modelling and Reformulating Constraint Satisfaction Problems p. 89 (2005)
103. Martınez-Hernández, B., Frisch, A.M.: The automatic generation of redundant representations and channelling constraints. In: Proceedings of the 5th International Workshop on Constraint Modelling and Reformulation. pp. 42–56 (2006)
104. Mears, C., De La Banda, M.G.: Towards automatic dominance breaking for constraint optimization problems. In: IJCAI. pp. 360–366 (2015)
105. Mears, C., De La Banda, M.G.: Towards automatic dominance detection in constraint optimisation problems. In: ModRef15 (2015)
106. Miller, H.E., Pierskalla, W.P., Rath, G.J.: Nurse scheduling using mathematical programming. Operations Research **24**(5), 857–870 (1976)
107. Mitchell, D.: A logical view of constraint modelling and reformulation. In: ModRef17 (2015)

108. Monette, J.N., Flener, P., Pearson, J.: Automated auxiliary variable elimination through on-the-fly propagator generation. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 313–329. Springer (2015)
109. Müller, T.: Solving set partitioning problems with constraint programming. In: PAPPACT 1998. pp. 313–332. The Practical Application Company, London, UK (March 1998)
110. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007), the MiniZinc toolchain is available at https://www.minizinc.org
111. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007. pp. 529–543. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
112. Nguyen, V.H.: Sat encodings of finite-csp domains: A survey. In: Proceedings of the Eighth International Symposium on Information and Communication Technology. pp. 84–91. SoICT 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3155133.3155167, http://doi.acm.org/10.1145/3155133.3155167
113. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving sat encoding of constraint problems through common subexpression elimination in savile row. In: International Conference on Principles and Practice of Constraint Programming. pp. 330–340. Springer (2015)
114. Nina, N.: Reformulation of global constraints. Ph.D. thesis, University of New South Wales, Australia (2011)
115. Nyberg, A., Westerlund, T., Lundell, A.: Improved discrete reformulations for the quadratic assignment problem. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 193–203. Springer (2013)
116. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. Tech. Rep. APES-56-2003, APES Research Group (January 2003), http://www.dcs.st-and.ac.uk/~apes/apesreports.html
117. Pierre Flener: Topic 1: Introduction (2018), available from http://user.it.uu.se/~pierref/courses/COCP/slides/T01-Introduction.pdf
118. Pierre Flener: Topic 4: Modelling (2018), available from http://user.it.uu.se/~pierref/courses/COCP/slides/T04-Modelling.pdf
119. Pierre Flener: Topic 7: Solving technologies (2018), available from http://user.it.uu.se/~pierref/courses/COCP/slides/T07-SolvingTechs.pdf
120. Prud'homme, C., Fages, J.G., Lorca, X.: Choco: A Free and Open-Source Java Library for Constraint Programming (2014), available at http://www.choco-solver.org
121. Puget, J.F.: On the satisfiability of symmetrical constrained satisfaction problems. In: Komorowski, J., Raś, Z. (eds.) ISMIS 1993. LNAI, vol. 689, pp. 350–361. Springer (1993)
122. Puget, J.F.: Symmetry breaking revisited. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 446–461. Springer (2002)
123. Puget, J.F.: A c++ implementation of clp (1994)
124. Régin, J.C.: Modeling problems in constraint programming. In Principles and Practice of Constraint Programming (CP) (2004)
125. Rendl, A.: Effective compilation of constraint models. Ph.D. thesis, University of St Andrews (2010)

126. Roney-Dougal, C.M., Gent, I.P., Kelsey, T., Linton, S.: Tractable symmetry breaking using restricted search trees. In: de Mántaras, R.L., Saitta, L. (eds.) ECAI 2004. pp. 211–215. IOS Press (2004)
127. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
128. Rousseau, L.M., Pesant, G., Gendreau, M.: A general approach to the physician rostering problem. Annals of Operations Research **115**(1–4), 193–205 (2002)
129. Saldanha, R., Morgado, E.: Solving set partitioning problems with global constraint propagation. In: Pires, F.M., Abreu, S. (eds.) Progress in Artificial Intelligence. pp. 101–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
130. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: SP 2010. pp. 513–528. IEEE Computer Society (2010), KALUZA is available at `http://webblaze.cs.berkeley.edu/2010/kaluza/`
131. Scott, J.D.: Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types. Ph.D. thesis, Department of Information Technology, Uppsala University, Sweden (2016), available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311`
132. Scott, J.D., Flener, P., Pearson, J.: Bounded strings for constraint programming. In: Brodsky, A. (ed.) ICTAI 2013. pp. 1036–1043. IEEE Computer Society (2013)
133. Scott, J.D., Flener, P., Pearson, J.: Constraint solving on bounded string variables. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 375–392. Springer (2015)
134. Scott, J.D., Flener, P., Pearson, J.: Constraint solving with bounded string variables. In: Michel, L. (ed.) CP-AI-OR 2015. LNCS, vol. 9075, pp. 373–390. Springer (2015)
135. Selensky, E.: A reformulation of the bridge building problem as vehicle routing. Modelling and Reformulating Constraint Satisfaction Problems p. 132 (2003)
136. Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M.: Learning from learning solvers. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming. pp. 455–472. Springer International Publishing, Cham (2016)
137. Simonis, H.: Reformulation: A practical view. In: ModRef02 (2002)
138. Palahí i Sitges, M., et al.: Reformulation of constraint models into SMT. Ph.D. thesis, Universitat de Girona (2015)
139. Smith, B.: CSPLib problem 001: Car sequencing. `http://www.csplib.org/Problems/prob001`
140. Smith, B.M.: Modelling. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 11, pp. 375–404. Elsevier (2006)
141. Smith, B.M.: Comparing dual viewpoints in permutation problems. Constraint Modelling and Reformulation (ModRef'09) p. 147 (2009)
142. Soto, R.: Languages and Model Transformation in Constraint Programming. Ph.D. thesis, Université de Nantes (2009)
143. Spracklen, P., Akgün, Ö., Miguel, I.: Automatic generation and selection of streamlined constraint models via monte carlo search on a model lattice. In: International Conference on Principles and Practice of Constraint Programming. pp. 362–372. Springer (2018)
144. Stuckey, P.J., de la Banda, M.G., Maher, M., Marriott, K., Slaney, J., Somogyi, Z., Wallace, M., Walsh, T.: The g12 project: Mapping solver independent models to efficient solutions. In: International Conference on Logic Programming. pp. 9–13. Springer (2005)

145. Stuckey, P.J., de la Banda, M.G., Maher, M., Marriott, K., Slaney, J., Somogyi, Z., Wallace, M., Walsh, T.: The g12 project: Mapping solver independent models to efficient solutions. In: Gabbrielli, M., Gupta, G. (eds.) Logic Programming. pp. 9–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
146. Swearngin, A., Choueiry, B.Y., Freuder, E.C.: A reformulation strategy for multi-dimensional csps: The case study of the set game. In: SARA (2011)
147. Tack, G.: Constraint propagation-models, techniques, implementation. Ph.D. thesis, Saarland University (2009)
148. Torres, J., Baier, J.A.: Polynomial-time reformulations of ltl temporally extended goals into final-state goals. In: Proceedings of the 24th International Conference on Artificial Intelligence. pp. 1696–1703. IJCAI'15, AAAI Press (2015), http://dl.acm.org/citation.cfm?id=2832415.2832485
149. Trick, M.: Formulations and reformulations in integer programming. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 366–379. Springer (2005)
150. Van Hentenryck, P.: The OPL Optimization Programming Language. The MIT Press (1999)
151. Van Hentenryck, P.: Constraint and integer programming in OPL. INFORMS Journal on Computing **14**(4), 345–372 (2002)
152. Walsh, T.: CSPLib problem 019: Magic squares and sequences. http://www.csplib.org/Problems/prob019
153. Walsh, T.: Sat v csp. In: International Conference on Principles and Practice of Constraint Programming. pp. 441–456. Springer (2000)
154. Walsh, T.: Permutation problems and channelling constraints. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 377–391. Springer (2001)
155. Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 650–664. Springer (2006)
156. Weld, D.S.: Approximation reformulations. In: AAAI. pp. 407–412 (1990)
157. Wetter, J., Akgün, Ö., Miguel, I.: Automatically generating streamlined constraint models with essence and conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 480–496. Springer (2015)
158. Williams, H.P.: Integer programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Model Building in Mathematical Programming, chap. 8, pp. 155–164. Wiley, fifth edn. (2013)
159. Williams, H.P.: Model Building in Mathematical Programming. Wiley, fifth edn. (2013)
160. Wolsey, L.A.: Building integer programming models. In: Integer Programming, chap. 3, pp. 263–400. Wiley (1998)
161. Yorke-Smith, N., Gervet, C.: Tight and tractable reformulations for uncertain csps. Proceedings of Constraint Programming (CP) **4** (2004)
162. Zanarini, A., Van Hentenryck, P.: Identifying patterns in sequences of variables. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 246–251. Springer (2011)
163. Zeighami, K., Leo, K., Tack, G., de la Banda, M.G.: Towards semi-automatic learning-based model transformation. In: International Conference on Principles and Practice of Constraint Programming. pp. 403–419. Springer (2018)
164. Zhou, J.: A constraint program for solving the job-shop problem. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 510–524. Springer (1996)
165. Zhou, J.: A permutation-based approach for solving the job-shop problem. Constraints **2**(2), 185–213 (1997)