

Evaluating ASP and commercial solvers on the CSPLib

Toni Mancini*, Davide Micaletto, Fabio Patrizi, Marco Cadoli

Dipartimento di Informatica e Sistemistica,
Università di Roma “La Sapienza”,
via Salaria 113, I-00198 Roma, Italy,
tmancini|micaletto|patrizi|cadoli@dis.uniroma1.it

Abstract

This paper¹ deals with four solvers for combinatorial problems: the commercial state-of-the-art solver ILOG OPLSTUDIO, and the research ASP systems DLV, SMOBELS and CMOBELS. The first goal of this research is to evaluate the relative performance of such systems when used in a purely declarative way, using a reproducible and extensible experimental methodology. In particular, we consider a third-party problem library, i.e., the CSPLib, and uniform rules for modelling and instance selection. The second goal is to analyze the marginal effects of popular reformulation techniques on the various solving technologies. In particular, we consider structural symmetry breaking, the adoption of global constraints, and the addition of auxiliary predicates. Finally, we evaluate, on a subset of the problems, the impact of numbers and arithmetic constraints on the different solving technologies.

Results show that there is not a single solver winning on all problems, and that reformulation is almost always beneficial: symmetry-breaking may be a good choice, but its complexity has to be carefully chosen, by taking into account also the particular solver used. Global constraints often, but not always, help OPL, and the addition of auxiliary predicates is usually worth, especially when dealing with ASP solvers. Moreover, interesting synergies among the various modelling techniques exist.

1 Introduction

The last decade has witnessed a large effort in the development of solvers for combinatorial problems. The traditional approach based on writing *ad hoc* algorithms, complete or incomplete, or translating in a format suitable for Integer

*Contact author.

¹A preliminary version of this paper appeared as [4].

Programming solvers², has been challenged by the use of libraries for Constraint Programming (CP), such as ILOG SOLVER³, interfaced through classical programming languages, e.g. C++ or Prolog. At the same time, the need for a higher level of abstraction and declarativeness led to the design and development of general purpose languages and systems for constraint modelling and programming –e.g. AMPL [16], OPL [36], XPRESS^{MP4} or GAMS [7]– and languages based on specific solvers, such as DLV [20], SMOELS [26], CMOELS [21], ASSAT [22], or LINGO [14].

The availability of such declarative languages has been, of course, a major step ahead towards the ripening of the constraint programming paradigm: by giving greater emphasis to the problem modelling task, the user is in principle relieved from the responsibility of dealing with many procedural, algorithmic, and technological aspects. All such issues are ideally left to the system.

On the other hand, with these languages, it is now the *quality* of the problem model that plays a fundamental role in solvers' efficiency: it is in fact very well-known that the most simple, intuitive, and straightforward formulations for a given problem rarely are the most efficient ones, and several techniques have been proposed in the literature in order to reformulate the original model into equivalent ones that are more efficiently evaluable by the solver at hand. Such techniques include symmetry-breaking (cf., e.g., [9, 28, 15, 23]), abstraction of constraints and variables (cf., e.g., [18, 12, 17, 2]), addition of implied constraints and of auxiliary variables (cf., e.g., [33, 38]), and use of global constraints (cf., e.g., [30]).

The afore-mentioned techniques have been often dealt with in the literature on a *per problem* basis, and only little work has been performed in order to understand how much they are expected to perform on a new problem, when using a given solving technology. Arguably, this knowledge, coupled with theoretical studies that generalize and standardize such reformulation techniques, is fundamental in order to build declarative constraint programming systems that automatically tweak the user's models in order to boost their performance. This is mandatory in order to provide the users with the possibility of writing simple and declarative constraint models for their problems, without dramatically loosing in efficiency, and is ultimately an obliged step in order to spread the use of the constraint programming paradigm towards non-specialists.

Such evolution is actually not new: a similar process has been experienced during the past decades in the development of languages and systems for querying databases: without the existence of effective tools for query preprocessing and optimization (cf., e.g., [35]), the relational model and modern query languages like SQL would improbably spread through real applications.

With such a long-term goal in mind, in previous work [2, 23, 24, 3] we showed how problem models can be regarded as logical formulae, how several of the afore-mentioned techniques for model reformulation may be generalized and

²e.g. ILOG CPLEX, cf. <http://www.ilog.com/products/cplex>.

³cf. <http://www.ilog.com/products/solver>.

⁴cf. <http://www.dashoptimization.com>.

highly standardized, and how automated tools can be practically used in order to check their applicability for the problem at hand. This led to the possibility, by the solving engine, to effectively perform specification-level analysis of the user’s model, and automatize several reformulation tasks.

In this paper we consider four well-known systems for constraint solving, and perform an extensive experimentation in order to answer the following main questions:

1. How do such systems perform when used in a purely declarative way?
2. Which are the effects of problem reformulation and other modelling aspects in their overall efficiency?

The last question poses, in turn, two more issues:

- 2.a. Which are the most promising reformulation techniques for any given solving technology?
- 2.b. How do the systems perform when two or more “good” reformulation techniques are used together?

The systems we consider are one Constraint Programming solver, i.e., ILOG OPLSTUDIO⁵, and three Answer Set Programming (ASP) solvers, namely DLV⁶, SMOBELS⁷, and CMOBELS⁸. ILOG OPLSTUDIO is a commercial state-of-the-art system for Constraint Programming that relies on the declarative language OPL [36] and on the well-known solvers CPLEX (linear) and SOLVER (backtracking-based, non linear) produced by ILOG. OPLSTUDIO uses default search strategies based on dynamic variable and value ordering heuristics, in case no explicit procedure is given by the user. For simplicity, in the following we refer both to the language and the implementation as OPL.

As for ASP instead, it emerged in the late ‘90s as a new logic programming paradigm, having its roots in non-monotonic reasoning, deductive databases and logic programming with negation as failure. Much research has been carried out in the last years in order to develop efficient and purely declarative solvers based on this semantics, and several systems, mainly produced in universities, are nowadays available. Our choices were directed to some of the most well-known solvers in this area. However, very promising competitors have recently been released, e.g., CLASP⁹, that deserve much attention. Interestingly, many ASP solvers, among which SMOBELS and CMOBELS, share the specification language through the common parser LPARSE¹⁰. Finally, while DLV and SMOBELS use proprietary solving engines, CMOBELS compiles ASP specifications, when given together with an instance, into an instance of the Propositional Satisfiability

⁵cf. <http://www.ilog.com>.

⁶cf. <http://www.dbai.tuwien.ac.at/proj/dlv/>.

⁷cf. <http://www.tcs.hut.fi/Software/smodels/>.

⁸cf. <http://www.cs.utexas.edu/users/tag/cmodels.html>.

⁹cf. <http://www.cs.uni-potsdam.de/clasp>.

¹⁰cf. <http://www.tcs.hut.fi/Software/smodels>.

problem (SAT). This last issue is an important argument for the inclusion of CMODELS in our experiments, since it allows to compare Constraint Programming with the fertile research on SAT, which produced many efficient solvers (cf., e.g., [19] and <http://www.satcompetition.org> for the recent impressive advances).

At the higher level, the systems considered exhibit interesting differences, including availability (OPL and ASP solvers are, respectively, payware and free-ware systems, the latter often being open source), paradigm (resp. algebraic and rule-based with minimal model semantics) and expressiveness of the modelling language (e.g., availability of arrays of finite domain variables vs. boolean matrices), compactness of constraint representation (e.g., availability of global constraints).

In order to answer the questions stated above, we present a reproducible and extensible experimental methodology. In particular, we consider a large third-party problem library, i.e. the CSPLib¹¹, very well-known in the Constraint Programming community, and uniform rules for modelling and instance selection. As for the reformulation techniques subject of our experiments, we consider *symmetry breaking*, the use of *global constraints*, and that of *auxiliary variables and predicates*. Moreover, we try to assess how negatively *numbers and arithmetics* influence performance.

In this way, we aim to evaluate the *marginal impact* that each issue has on the performance of the different solvers. The significance of the experiments is achieved by considering a large set of problems and a high number of instances. As a side-effect, we also aim to advance the state of knowledge on the good practices in modelling for some important classes of solvers.

Comparison among different solvers for CP has already been addressed in the literature: in particular, we recall [13] and [37] where SOLVER is compared to other systems such as, e.g., OZ [34], CLAIRE¹², and various Prolog-based engines. Moreover, some benchmark suites have been proposed, cf., e.g., COCONUT [31]. Also on the ASP side, which has been the subject of much research in the recent years, benchmark suites have been built in order to facilitate the task of evaluating improvements of their latest implementations, the most well-known being ASPARAGUS¹³ and ASPLib¹⁴. However, less research has been done in comparing solvers based on different formalisms and technologies, and in evaluating the relative impact of different features and modelling techniques. In particular, very few papers compare ASP solvers to state-of-the-art systems for CP. To this end, we cite [11], where two ASP solvers are compared to a CLP(FD) Prolog library on six problems: Graph coloring, Hamiltonian path, Protein folding, Schur numbers, Blocks world, and Knapsack, and [27], where ASP and Abductive Logic Programming systems, as well as a first-order finite model finder, are

¹¹cf. <http://www.csplib.org>.

¹²cf. <http://claire3.free.fr>.

¹³cf. <http://asparagus.cs.uni-potsdam.de>.

¹⁴cf. <http://dit.unitn.it/~wasp>.

compared in terms of modelling languages and relative performances on three problems: Graph coloring, N-queens, and a scheduling problem.

Outline. The outline of the paper is as follows: in Section 2 we present and discuss the adopted experimental methodology, while in Section 3 we describe the experimental framework, i.e., the selected problems, their various formulations, and how we chose their instances. Then, in Section 4 we present and analyze the results of our experiments. Finally, Section 5 concludes the paper.

2 Methodology

In this section we present the methodology adopted in order to achieve the goals mentioned in Section 1. For each problem, we define a number of different formulations: a *base specification*, obtained by a straightforward and intuitive “translation” of the CSPLib problem description into the target language, and several *reformulated* ones, obtained by using different techniques proposed in the literature: (i) symmetry-breaking, (ii) addition of global constraints and (iii) addition of auxiliary predicates. Moreover, in order to establish whether merging different reformulations, which are proven to improve performance when used alone, speeds-up even more the computation, we considered additional specifications for the same problems, obtained by *combining* the aforementioned techniques, and exploring the existence of synergies among them. Finally, an evaluation of the impact of numbers and arithmetic constraints in all the solvers involved in the experimentation has been performed on two problems.

The goals declared in Section 1 drive us to follow a purely declarative approach during modelling: hence, all the specifications have been solved without performing any kind of tuning of the search parameters and without specifying any procedural aspects (like, e.g., *ad hoc* search procedures in OPL). Thus, the experimentation relies on the *default behavior* of the different solvers (i.e., grounding techniques, branching heuristics, etc.). In fact, we remind that our purpose is not to solve a particular problem in the most efficient way as possible, but to understand what performance the various systems may offer to the modeller in a transparent way, i.e., without requiring him to take into account any details about the underlying technologies, and which is the effectiveness and the marginal impact of various high-level and strongly standardized reformulation techniques on solvers’ performance. However, the end of Section 4 presents some discussions about the price payed for this high level of declarativeness.

In order to achieve such objectives, the modelling task has been performed in a way as systematic as possible, by requiring the specifications of the various solvers to be *similar* to each other. The criteria followed during the modelling task are discussed in Section 3.2. As for the instances, in this paper we opted for problems whose input data is made of few integer parameters (with two exceptions, which are discussed in Section 3.1).

Due to the high number of problems and instances solved, the necessity of having a synthetic yet meaningful measure of the various solvers performance arises. Various approaches exist in the literature, e.g., comparing the overall solution time for a given set of instances to each problem, or the number of correctly solved instances in a given time limit (cf., e.g, [25]). However, due to the nature of the chosen problems (instances depending on very few integer parameters, straightforwardly denoting their size), we decided to follow a different approach: for each problem we fixed all the input parameters but one (the reasonability of such choices are discussed in Section 3.1), and in our results we report, for each problem and solver, the *largest instance* (denoted by the value given to the selected parameter) that is solvable in a given time limit (one hour) by each reformulation.¹⁵ For one single problem such approach was intrinsically infeasible (cf., discussion on the Water buckets problem in Section 3.1), and so we exceptionally built a set of benchmark instances for it, and compared the overall solving time needed by each solver to manage them. Another problem instead (Car sequencing, cf. discussion in the same section) needed some extra care in order for its instances to be described by a single parameter.

3 The experimental framework

3.1 Problems selection

In this research we consider the CSPLib problem library for our experiments. CSPLib is a collection of 45 problems and is one of the most widely known source of benchmarks in the CP community. Problems are classified into 7 areas: Scheduling, Design, Configuration and diagnosis, Bin packing and partitioning, Frequency assignment, Combinatorial mathematics, Games and puzzles, and Bioinformatics.

For our experiments we chose 10 problems from CSPLib, that cover all the 7 areas of the collection. Since many problems in CSPLib are described only in natural language, without any formal characterization, this work also provides, as a side-effect, their formal specifications in the modelling languages adopted by several solvers.

In what follows we give a description of the chosen problems, as well as their identification numbers in CSPLib, the applications areas they belong to, and the parameters chosen to define their instances.

Ramsey problem (#017: Bin packing and partitioning, Combinatorial mathematics). This problem amounts to color the edges of a complete graph with n nodes using the minimum number of colors, in order to avoid monochromatic triangles. Instances of the problem are denoted by the number of graph nodes, n .

¹⁵As discussed in Section 4, in order to neutralize some noise in the solvers' behavior, the definition of *largest solvable instance* has to be slightly emended.

Social golfer (#010: Scheduling, Bin packing and partitioning, Games & puzzles). In a golf club there are 32 *social* golfers who play once a week in 8 groups of 4. The problem amounts to find a schedule for as many as possible weeks, such that no two golfers play in the same group more than once. Here we consider a decisional version of the problem: given a positive integer w , the goal is to find a schedule for w weeks. Problem instances are thus denoted by the number of weeks w .

Golomb rulers (#006: Frequency assignment, Combinatorial mathematics). Given a positive integer m , this problem amounts to put m marks on a ruler, in such a way that the $m(m-1)/2$ distances among them are all different. The objective is to find the length of the shortest ruler that admits the positioning of m marks. Instances are denoted by values given to m .

Car sequencing (#001: Scheduling). A number of cars have 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.). Such 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 them. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded.

Selection of instances for this problem differs from the previous cases, since they cannot be directly encoded by a single parameter. We considered some benchmarks suggested in the CSPLib, namely “4/72”, “6/76” and “10/93”. However, since they were too hard for our solvers, we proceeded as follows, in order to generate a new set of (smaller) instances: from any original benchmark (with n classes), we generated a set of instances by reducing the number of classes to all possible smaller values $k \in [1, n]$. In particular, for each such k we generated an instance with k classes, also suitably resizing the original number of cars and station capacities, in order to preserve satisfiability. Thus, instances derived from the same benchmark could be ordered according to the value for the (reduced) number of classes k , which could then be regarded as a measure for their size.

Water buckets (#018: Design, Configuration and diagnosis, Bin packing and partitioning, Games & puzzles). We consider a generalization of the CSPLib specification, which is as follows: Given an 8 pint bucket of water, and two empty buckets which can contain 5 and 3 pints respectively, the problem requires to divide the water into two by pouring water between buckets (that is, to end up with 4 pints in the 8 pint bucket, and 4 pints in the 5 pint bucket) in the smallest number of transfers.

The generalization consists in making the specification parametric with respect to the start and goal configurations, which are now inputs to the problem.

We also designed 11 instances (each one denoted by a start and a goal configuration) from that considered in the original problem description, which have been proved to be non-trivial by preliminary experiments. Since such instances could not be denoted by a single parameter, solvers' performance for this problem and the different specifications have been compared by considering the overall time needed to solve them.

Maximum density still life (#032: Games & puzzles). This problem arises from the Game of life, invented by John Horton Conway in the 1960s (cf., e.g., [10] for a detailed description). Life is played on a squared board, considered to extend to infinity in all directions. Each square of the board is a cell, which at any time during the game is either alive or dead. A cell has eight neighbours. The configuration of alive and dead cells at time t leads to a new configuration at time $t + 1$ according to the rules of the game:

- If a cell has exactly three living neighbours at time t , it is alive at time $t + 1$;
- If a cell has exactly two living neighbours at time t , it is in the same state at time $t + 1$ as it was at time t ;
- Otherwise, the cell is dead at time $t + 1$.

A stable pattern, or still-life, is not changed by these rules. Hence, every cell that has exactly three alive neighbours is alive, and every cell that has fewer than two or more than three alive neighbours is dead. (An empty board is a still-life, for instance.)

Given a positive integer n , the problem amounts to find the densest possible still-life pattern, i.e. the pattern with the largest number of alive cells, that can be fitted into an $n \times n$ section of the board, with all the rest of the board containing dead cells. Instances are encoded by n , the size of the board.

Word design for DNA computing on surfaces (#033: Bioinformatics). This problem amounts to find as large a set S of strings (words) of length 8 over the alphabet $W = \{A, C, G, T\}$ with the following properties:

- Each word in S has 4 symbols from $\{C, G\}$;
- Each pair of distinct words in S differ in at least 4 positions; and
- Each pair of words x and y in S (where x and y may be identical) are such that $R(x)$ and $C(y)$ differ in at least 4 positions, where:
 - $R(x)$ is the reverse of the input string x ;
 - $C(y)$ is the Watson-Crick complement of y , i.e. the word obtained by y by replacing each A by a T and vice versa and each C by a G and vice versa.

Here we consider a decisional version of the problem: given a positive integer c , the goal is to find, if possible, a set of words S with the above properties, and such that $|S| = c$. Instances are encoded by c .

Magic squares (#019: Combinatorial mathematics, Games & puzzles). An order n magic square is a n by n matrix containing the numbers 1 to n^2 , with each row, column and main diagonal equal the same sum. Given a positive integer n , the problem amounts to find a n order magic square. Instances are encoded by n , the matrix order.

Langford numbers (#024: Combinatorial mathematics, Games & puzzles). This is a generalization of the specification given in the CSPLib (which fixes the forthcoming value n to 4). Given two sets of the numbers from 1 to n , the problem amounts to arrange the $2n$ numbers in the two sets into a single sequence in which the two 1's appear one number apart, the two 2's appear two numbers apart, the two 3's appear three numbers apart, ..., and the two n 's appear n numbers apart. Instances are encoded by n .

All-interval series (#007: Frequency assignment, Combinatorial mathematics). Given the twelve standard pitch-classes (c , $c\#$, d , ...), represented by numbers $0, 1, \dots, 11$, this problem amounts to find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). That is, for each of the intervals, there is a pair of neighboring pitch-classes in the series, between which this interval appears.

We consider a generalization of this problem in which the set of numbers is the range from 0 to $n - 1$, for any given positive n . In particular, given such n , the problem amounts to find a vector $s = (s_1, \dots, s_n)$ that is a permutation of $\{0, 1, \dots, n - 1\}$ and such that the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n - s_{n-1}|)$ is a permutation of $\{1, 2, \dots, n - 1\}$. Instances are encoded by the integer n .

Table 1 summarizes the set of problems considered in this paper, as well as the parameters used to encode their respective instances.

Some comments on the choices made above are in order. First of all, since the performance typically depends on the instances being positive or negative (i.e., satisfiable or unsatisfiable), we considered 4 *optimization* problems. As a matter of fact, for proving that a solution is optimal, solvers have to solve both positive and negative instances.

Actually, two more problems, namely Social golfer and Word design, are presented as optimization problems in the CSPLib. However, since none of our solvers were able to achieve their optimal solutions,¹⁶ in our experiments we considered their decisional versions.

¹⁶We observe that, as for Social golfer, the longest schedule is still unknown in the literature, being of 9 or 10 weeks.

| Id | Problem name | Instances defined by |
|-----|------------------------------------|----------------------|
| 017 | Ramsey | # of graph nodes |
| 010 | Social golfer (decisional version) | Schedule length |
| 006 | Golomb rulers | # of marks |
| 001 | Car sequencing | # of classes |
| 018 | Water buckets | Benchmark instances |
| 032 | Maximum density still life | Board size |
| 033 | Word design (decisional version) | # of words |
| 019 | Magic squares | matrix order |
| 024 | Langford numbers | # of values |
| 007 | All-interval series | # of integers |

Table 1: The set of problems considered in the experiments, together with their CSPLib identification number and the unique parameter used to encode their instances. We observe that instances for two problems could not naturally be encoded by a single parameter: in these cases, benchmarks available (or derived from those available) on CSPLib were used.

The choice of the problems had also to take into account that one of the solvers, CMODELS, is unable to natively deal with optimization problems. In order to let the experimentation being significant for such solver, we proceeded to solve the 4 optimization problems with CMODELS by iteratively solving instances of their decisional versions, in order to find the best value of the objective function through binary search (this issue is discussed in detail in Section 4). Of course, the performance obtained by CMODELS in this way may be considered as a lower bound of that which can be obtained by extending the system in order to natively tackle optimization problems. However, experimental results show that even in this case, this solver is competitive with respect to (and sometimes even faster than) the others.

As for the choice of the instances, those of two problems, namely Car sequencing and Water buckets, could not be naturally encoded by a single parameter. Hence, they have been derived from benchmarks taken from the CSPLib.

Finally, some problems, namely Water buckets and Magic squares have been used also to evaluate the impact of numbers and arithmetic constraints in problem models. In fact, it is well known that such issues may greatly degrade solvers' performance, and that this behavior strongly depends on the underlying solving algorithm. In order to understand how negatively numbers and arithmetic constraints affect the behavior of the various solvers, we built new sets of instances, equivalent to the original ones, that however force the underlying algorithms to deal with larger numbers. By measuring and comparing how much their performance degrades, we are able to evaluate the robustness of the various technologies with respect to such issues.

3.2 Models selection

As claimed in Section 2, in order to build an extensible experimental framework, we followed the approach of being as uniform and systematic as possible during the modelling phase, by requiring the specifications of the various solvers to be similar to each other. Hence, even if not always identical because of the intrinsic differences among the languages that could not be overcome, all of the models share the same ideas behind the search space and constraints definitions, independently of the language. Below, we discuss the general criteria followed during the modelling phase, and the different formulations considered for each problem. Encodings of all problems for all solvers are available in the web-appendix of this paper.¹⁷ We refer the reader to it for a detailed understanding of the different problems' formulations described in what follows.

3.2.1 General modelling criteria

The first obvious difference between OPL and the ASP solvers concerns the search space declaration. The former relies on the notion of *function* from a finite domain to a finite codomain, while the latter ones have just *relations*, which must be restricted to functions through specific constraints. Domains of relations can be implicit in DLV, since the system infers them by contextual information. For each language, we used the most natural declaration construct, i.e, functions for OPL, and untyped relations for ASP. Secondly, since the domain itself can play a major role in efficiency, sometimes it has been inferred through some *a posteriori* consideration. This is especially the case for domains of objective values in optimization problems. As an example, in Golomb rulers the maximum position for marks (hence the rule length to be minimized) is upper-bounded by 2^m (m being the number of marks), but choosing such a large number can advantage OPL, which has powerful arc-consistency algorithms for domain reduction. As a consequence, we used the upper bound $3L/2$ for all solvers, L being the maximum mark value for the optimum of the specific instance. Such practice has been carried out systematically for each minimization problem (3 out of 4), by fixing the upper bound for the value of the objective function to $3L/2$, with L being the optimum (computed by preliminary experiments). As for the unique maximization problem considered (Maximum density still life), as discussed in Section 4 (cf. also the problem description given in Section 3.1), this issue does not apply.

3.2.2 Base specifications

The first formulation considered for each problem is the so called *base specification*. This has been obtained by a straightforward translation of the CSPLib problem description into the target language, by taking into account the general criteria discussed above, and, arguably, is the most natural and declarative.

¹⁷ cf. <http://www.dis.uniroma1.it/~tmancini/index.php?currItem=research-publications.webappendices.csplib2x>.

Of course, in general, different formulations may exist for a given problem that could alternatively be considered as its base specifications, since they are equally “natural” and “straightforward”. An example is given by the Water buckets problem, where elements of the search space (i.e., plans) could either be modelled as sequences of buckets *configurations* (e.g., for 3 buckets, we can explicitly maintain, for each time-point, the amount of water in each of them), or as sequences of *actions* that encode transitions from one configuration to another. To give the intuition, consider an instance with 3 buckets having capacity, respectively, 8, 5, 3, and which initial state being $\langle 8, 0, 0 \rangle$ (i.e., the first bucket is full, while the other two are empty). The following sequence of states is a (partial) plan consistent with the problem constraints, hence a point of the search space:

$$\langle \langle 8, 0, 0 \rangle, \langle 3, 5, 0 \rangle, \langle 3, 2, 3 \rangle, \dots \rangle,$$

meaning that at time-point 0 the buckets are in the initial configuration, at time-point 1, bucket 1 contains 3 units of water, bucket 2 contains 5, and so on. This plan could equivalently be encoded by representing transitions that link each state to the next one, i.e.:

$$\langle 1 \rightarrow 2, 2 \rightarrow 3, \dots \rangle,$$

meaning that the configuration at time-point 1 is obtained by moving water from bucket 1 to bucket 2, that at time-point 2 is in turn achieved by further moving water from bucket 2 to bucket 3, and so on.

As a general rule, in all these cases, we performed preliminary experiments in order to understand which formulation led to the best performance, and regarded that one as the base specification.

3.2.3 Reformulation by symmetry-breaking

It is well known in the literature (cf. references given in Section 1) that the symmetries exhibited by a problem can be one of the major sources of inefficiency for many classes of solvers. Hence, the first kind of reformulation considered aims at evaluating the impact of performing symmetry-breaking.

Given the high abstraction level of the languages, an immediately usable form of reformulation is through the addition of new constraints (cf., e.g., [9, 28, 15]), usually called *symmetry-breaking constraints*.

Symmetry-breaking is dealt with in a systematic way, by adding to the base specifications general, uniform, and standardized schemas for symmetry-breaking presented in [23]. Such schemas are briefly recalled in what follows (examples below are given in the simple case where all permutations of values are symmetries; generalizations exist):

- **Selective assignment (SA):** A subset of the variables are assigned to precise domain values. As an example, in the Social golfer problem, in order to break the permutation symmetries among groups, we can fix the group assignment for the first and partially for the second week.

- **Selective ordering (SO):** Values assigned to a subset of the variables are forced to be ordered. As an example, in the Golomb rulers problem, in order to break the symmetry that “reverses” the ruler, we can force the distance between the first two marks to be less than the difference between the last two.
- **Lowest-index ordering (LI):** Linear orders are fixed among domain values and variables, and assignments of variables (x_1, \dots, x_n) are required to be such that, for any pair of values (d, d') , if $d < d'$ then $\min\{i | x_i = d\} < \min\{i | x_i = d'\}$. An example is given by the Ramsey problem: once orders are fixed over colors, e.g. red < green < blue, and over edges, we can force the assignments to be such that the least index of red colored edges is lower than the least index of green colored ones, and analogously for green and blue edges.
- **Size-based ordering (SB):** After fixing a linear order on values, assignments are forced to be such that $|\{x \in V | x = d\}| \leq |\{x \in V | x = d'\}|$, for any pair of values $d \leq d'$, V being the set of variables. As an example, in the Ramsey problem we could require the number of blue colored edges to be greater than or equal to that of green ones, in turn forcing the latter to be greater than or equal to the number of red colored edges. Generalizations of this schema do exist, depending on the way the partition of the variables set into size-ordered sets is defined.
- **Lexicographic ordering (LX):** This schema is widely applied in case of search spaces defined by matrices, where all permutations of rows (or columns) are symmetries. It consists in forcing the assignments to be such that all rows (resp. columns) are lexicographically ordered.
- **Double-lex (lex^2) ordering (L2):** This is a generalization of the previous schema, applicable where the matrix has both rows and columns symmetries. It consists in forcing assignments to be such that both rows and columns are lexicographically ordered (cf., e.g., [15]). An example is Social golfer, in which the search space can be defined as a 2D matrix that assigns a group to every player/week combination. Such a matrix has all rows and columns symmetries (we can swap the schedules of any two players, and the group assignments of any two weeks).

Above schemas for symmetry-breaking can be qualitatively classified in terms of “how much” they reduce the search space (i.e., their *effectiveness*), and in terms of “how complex” is their evaluation. In particular they can be partially ordered as follows: SA < SO < LI < LX < L2, and LI < SB, where $s_1 < s_2$ means that schema s_2 better reduces the search space. However, s_2 typically requires more complex constraints than s_1 . Figure 1 shows such a classification as a partial order set.

Furthermore, in many cases, more than a single schema is applicable for breaking the symmetries of a given specification. Since previous studies [29]

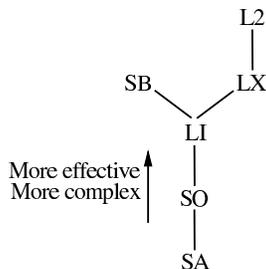


Figure 1: Classification among symmetry-breaking schemas as a partial ordered set. Higher schemas are more effective, but usually require more complex constraints.

showed that this technique is effective when simple formulae are added, it is interesting to know –for each class of solvers– what is the amount of symmetry breaking that can be added to the model, and still improving performance. In what follows, we give a partial answer to this question.

3.2.4 Reformulation by exploiting global constraints

Global constraints (GC) encapsulate, and are logically equivalent to a set of other constraints. Despite this equivalence, global constraints come with more powerful filtering algorithms, and a specification exhibiting them is likely to be much more efficiently evaluable.

One of the most well-known global constraints supported by constraint solvers is `alldifferent(x1, . . . , xn)` that forces the labeling algorithm to assign different values to all its input variables. Of course, such a constraint can be replaced by a set of binary inequalities $x_i \neq x_j$ (for all $i \neq j$), but such a substitution will result in poorer propagation, hence in less efficiency.

Several global constraints are supported by OPL, e.g., `alldifferent` and `distribute`. The latter is a generalization of `alldifferent`, forcing the algorithm to assign a set of values to a set of variables in such a way that any value occurs a given number of times among the variables.

According to the problems structure, `alldifferent` has been applied to Golomb rulers, Magic squares, and All-interval series, and `distribute` to Social golfer, Car sequencing, Word design, and Langford problem. As for Ramsey, Water buckets, and Maximum density still life, none of such reformulations applies.

On the other hand, ASP solvers do not offer this kind of global constraints,¹⁸ hence no comparison can be made on this issue.

¹⁸Although they provide various aggregate and weight constructs, which may be regarded as a special form of global constraints.

3.2.5 Reformulation by adding auxiliary predicates

A guessed predicate (i.e., a predicate belonging to the search space declaration) is called *auxiliary* if its extensions functionally depend on those of the other ones. The use of auxiliary guessed predicates is very common in declarative programming, especially when the user needs to store partial results, to maintain intermediate states, or wants to make the so-called *redundant modelling* (cf., e.g., [8]), by maintaining multiple views of the search space, synchronized by channelling constraints.

As an example, consider the specification of Social golfer, where the search space is a total function assigning a group to each player/week pair. We could add to the search space an auxiliary guessed predicate $meet(\cdot, \cdot, \cdot)$ and suitable channelling constraints that, for each pair of players p_1, p_2 ($p_1 \neq p_2$), and each week w , force the triple $\langle p_1, p_2, w \rangle$ to belong to $meet$ iff players p_1 and p_2 are expected to play in the same group on week w , according to the guessed scheduling.

It can be observed that any extension of the guessed scheduling uniquely defines an extension for the $meet$ predicate (this is called a *functional dependence* among guessed predicates —cf., e.g., [24]). The main advantage of using such predicate is in the simplification of some of the constraints: as an example, the *meet-at-most-once* constraint can be much more compactly expressed in terms of the $meet$ predicate.

In general, although the use of auxiliary predicates increases —from a conceptual standpoint— the size of the search space, this often results in a simplification of complex constraints and in a reduction of the number of their variables, and thus may lead, as we show in Section 4, to appreciable time savings.

We consider equivalent specifications, obtained by using auxiliary predicates, for all of the selected problems. However, the bottom-up grounding techniques and evaluation algorithms of DLV and LPARSE (the grounder used by both SMODELS and CMODELS) may significantly advantage ASP solvers over OPL on such specifications, since auxiliary predicates are usually defined in rule-heads. To this end, when adding auxiliary predicates to OPL specifications, we also added simple, general, and high-level modelled search procedures instructing the labelling algorithm to delay branches on auxiliary variables, while maintaining the default behavior on the others, as explained in previous work [24].

3.2.6 Reformulation by combining different techniques

In many cases, more than one single reformulation strategy improves performance on a given problem. Hence, the question arises whether synergies exist among them, and what techniques are likely to work well together, for each solver.

To this end, for each problem we consider some additional formulations: the first one has been obtained, according to the aforementioned uniformity criteria, by merging the two reformulations (among symmetry-breaking, addition of global constraints and of auxiliary predicates) that, for each solver, resulted

to be the most efficient. Finally, in order to understand whether there exist better, undiscovered synergies, we relaxed the uniformity hypothesis, and considered some of the other possible combinations of reformulation strategies, with the goal to further boost performance.

3.2.7 Example: the Golomb rulers problem

In order to show how we applied our methodology to modelling and reformulation, we present the different formulations we designed for the Golomb rulers problem (cf. Section 3.1) in the various languages. The reader is invited to look at the web-appendix (cf. footnote 17) for all the other specifications.

Base specifications

OPL

```
int+ n_marks = ...;
int+ maxval = ...; // Fixed to 3L/2 (L = opt. length for n_marks marks)

var int+ ruler[1..n_marks] in 0..maxval; // search space decl.
minimize ruler[m] // obj. func.
subject to {
  ruler[1] = 0; // c1
  forall (i in 1..,n_marks-1) { ruler[i] < ruler[i+1]; }; // c2
  forall(i,j,k,l in 1..n_marks :
    (i < j) & (k < l) & (i <> k \ / j <> l)) { // c3
    (ruler[j] - ruler[i]) <> (ruler[l] - ruler[k]);
  };
};
```

DLV

```
ruler(P,M) v fail(P,M) :- marks(M), positions(P). % search space decl.
:- positions(P), not #count{ M : ruler(P,M)}=1.
ruler(1,0). % c1
:- ruler(X,Y), ruler(X1,Y1), X1>X, Y>=Y1. % c2
:- #count{X1,X2 : ruler(X1,Y1), ruler(X2,Y2), X2>X1,
  Y2=D+Y1} > 1, marks(D). % c3
:~ ruler(P,M), n_marks(P). [M:1] % obj. func.
```

SMODELS and CMODELS (LPARSE)

```
1 {ruler(P,M): marks(M)} 1 :- pos(P). % search space decl.
compute {ruler(1,0)}. % c1
:- ruler(X,Y), ruler(X1,Y1), gt(X1,X), not gt(Y1,Y). % c2
:- ruler(X,Y), ruler(X1,Y1), ruler(X2,Y2), % c3
  ruler(X3,Y3), gt(X1,X), gt(X3,X2),
  or(neq(X,X2),neq(X1,X3)), eq(Y1-Y,Y3-Y2).
minimize[ruler(n_marks,Y): marks(Y)]. % obj func.
```

Reformulation by symmetry-breaking: Selective ordering

```
OPL

// ...Base specification plus:
((ruler[2] - ruler[1]) < (ruler[n_marks] - ruler[n_marks - 1])); // S0

DLV

% ...Base specification plus:
:- ruler(1,Y), ruler(2,Y1), ruler(M,Y3), ruler(N,Y2),           % S0
   Y1=D1+Y, Y3=D2+Y2, D2<D1, M=N+1, n_marks(M).

SMODELS and CMODELS (LPARSE)

% ...Base specification plus:
:- ruler(1,Y), ruler(2,Y1), ruler(n_marks-1,Y2),               % S0
   ruler(n_marks,Y3), lt(Y1-Y,Y3-Y2).
```

Reformulation by exploiting global constraints: all-different

```
OPL

// ...Base specification, with c3 replaced by:
alldifferent(
  all (i,j in 1..n_marks: i <> j) (ruler[j] - ruler[i]));      // c3 (GC)
```

DLV, SMODELS, and CMODELS: not applicable.

Reformulation by adding auxiliary predicates

```
OPL

int+ n_marks = ...;
int+ maxval = ...;
int+ numOfDifferences = (n_marks*(n_marks-1))/2;
range diffValues 1..maxval;

var int+ ruler[1..n_marks] in 0..maxval;           // search space decl.
var diffValues distance[1..numOfDifferences];     // aux. pred. decl.

minimize ruler[n_marks]                           // obj. func.
subject to {
  ruler[1] = 0;                                     // c1
  forall (i in 1..n_marks-1) {                     // c2
    ruler[i] < ruler[i+1];
  }
}
```

```

};
forall(m1,m2 in 1..n_marks: m2 > m1){           // chann. constr.
    distance[((n_marks*(n_marks-1)/2) -
              ((n_marks-m1+1)*(n_marks-m1)/2) + (m2-m1))] =
        ruler[m2] - ruler[m1]
};
forall(i,j in 1..numOfDifferences : i > j) {     // c3 (Aux)
    distance[i] <> distance[j];
};
};

// Delay branches on aux. predicate
search{
    generate(ruler);
    generate(distance);
};

DLV

ruler(P,M) v fail(P,M) :- marks(M), positions(P). % search space decl.
:- positions(P), not #count{ M : ruler(P,M)}=1.

ruler(1,0).                                     % c1
:- ruler(X,Y), ruler(X1,Y1), X1>X, Y>=Y1.      % c2
diff(X1,X2,D) :- ruler(X1,Y1), ruler(X2,Y2),    % aux. pred. decl. and chann. constr.
                X2>X1, Y2=D+Y1.
:- #count{X1,X2: diff(X1,X2,D)}>1, marks(D), D>0. % c3 (Aux)
:~ ruler(P,M), n_marks(P). [M:1]                % obj. func.

SMODELS and CMODELS (LPARSE)

1 {ruler(P,M): marks(M)} 1 :- pos(P).           % search space decl.
compute {ruler(1,0)}.                           % c1
:- ruler(X,Y), ruler(X1,Y1), gt(X1,X), not gt(Y1,Y). % c2
diff(X1,X2,D) :- ruler(X1,Y1), ruler(X2,Y2),    % aux. pred. decl. and chann. constr.
                gt(X2,X1), D=Y2-Y1.

:- 2{diff(A,B,D): positions(A;B): gt(B,A)}, marks(D). % c3 (Aux)
minimize[ruler(n_marks,Y): marks(Y)].           % obj func.

```

A combined reformulation We give the OPL specification obtained combining symmetry-breaking (SO), adoption of global constraints (all-different), and addition of the auxiliary predicate:

```

int+ n_marks = ...;
int+ maxval = ...;
int+ numDifferences = (n_marks*(n_marks-1)/2);
range diffValues 1..maxval;

var int+ ruler[1..n_marks] in 0..maxval;           // search space decl.
var diffValues distance[1..numDifferences];       // aux. pred. decl.

minimize ruler[n_marks]                          // obj. func.
subject to {
  ruler[1] = 0;                                   // c1
  forall (i in 1..n_marks-1) {                   // c2
    ruler[i] < ruler[i+1];
  };
  forall(m1,m2 in 1..n_marks: m2 > m1){         // chann. constr.
    distance[((n_marks*(n_marks-1)/2) -
      ((n_marks-m1+1)*(n_marks-m1)/2) + (m2-m1))] =
      ruler[m2] - ruler[m1]
  };
  alldifferent(distance);                         // c3 (Aux+GC)
  ((ruler[2] - ruler[1]) <                       // S0
    (ruler[n_marks] - ruler[n_marks - 1]));
};

// Delay branches on aux. predicate
search{
  generate(ruler);
  generate(distance);
};

```

4 Experimental results

In this section we present and analyze the results of our experimentation. Experiments have been performed by using the following solvers:

- ILOG OPLSTUDIO 3.61, that invokes ILOG SOLVER v. 5.3 as search engine,¹⁹
- DLV v. 2005-02-23,
- SMODELS v. 2.28, by using LPARSE 1.0.17 for grounding,
- CMODELS v. 3.55, by using LPARSE 1.0.17 for grounding,

on a 2 CPU Intel Xeon 2.4 Ghz computer, with a 2.5 GB RAM and Linux v. 2.4.18-64GB-SMP.

¹⁹Since specifications for all of our problems are non-linear, we made no use of the linear engine CPLEX.

As discussed in Sections 2 and 3.2, for every problem, we wrote several specifications (a base plus several reformulated ones), in the different languages, having care to be as uniform and systematic as possible during the modelling phase. We then ran the different specifications for each solver on the same set of instances, which selection has been performed as discussed in Section 3.1. All runs had a timeout of 1 hour.

Results are shown in Tables 2 and 3 for, respectively, optimization and decision problems. In particular, for each problem and solver, we report the *largest instance* the various systems were able to solve (in the given time-limit) for the base specifications and the various reformulations.

However, during our experiments, we experienced that, for a given specification, sometimes solvers were unable to solve instance denoted by value k of the parameter in the time-limit, but were able to solve the one denoted by $k + 1$. This “noisy” (or “non-monotone”) behavior is very well-known in the literature. To ensure fairness, and to avoid the effects of such a noisy behavior, in the results *we intend for largest instance* the one denoted by the largest value k of the parameter that satisfies the following three conditions: (i) It is solvable in the time-limit by the given solver with the given specification; (ii) Neither instance $k + 1$ nor instance $k + 2$ are solvable in the time-limit by the same solver with the same specification; (iii) Do not exist instances k' and $k' + 1$ ($k' \leq k - 2$) which are not solvable in the time-limit by the same solver with the same specification.

We also remind that one solver, namely CMODELS, cannot natively deal with optimization problems. Nonetheless, CMODELS has been run also to solve the instances of the 4 optimization problems: its entries in Table 2 have been obtained by iteratively invoking the solver several times, by adding each time new constraints to the specifications that limit the values of the objective function evaluated on the solutions found, by following the approach of performing a binary search of the optimum objective value. Of course, such results (denoting the largest instances for which the overall time needed to find their optimum solutions was below the time-limit) represent lower bounds of the solver’s performance. Nonetheless they show that, even in this case, CMODELS is competitive with respect to the other solvers, thus confirming that SAT should be undoubtedly regarded as one of the most promising and effective technologies for constraint problems to date (such evidence has also been discussed in related work [5]). Figure 2 shows the algorithm used to solve minimization problems (3 out of 4) with CMODELS. As for the unique maximization problem, i.e., Maximum density still life, we observe that a dual approach could be straightforwardly used, since maximizing the number of alive cells does not increase the size of the search space (the total number of variables remains unchanged).

As discussed in Section 3.1, the size of the instances for few problems could not be easily represented by a single parameter. To this end, as for Car sequencing, we report, for each set of instances generated from CSPLib benchmarks “4/72”, “6/76” and “10/93” (cf. the discussion about instance selection for this problems

in Section 3.1), the largest one solved, i.e., the one with the largest number of classes, and, as for Water buckets, the overall time needed to solve the whole set of instances (instances that could not be solved contributed with 3,600 seconds to the overall time).

Finally, as for Langford numbers, it is known that positive instances are only those with $n = 4k$ or $n = 4k - 1$ ($k \in \mathbb{N}$). To this end, in Table 3 we separately report the largest size of the positive and negative instances solved.

From these results, the following observations about the relative performance of the various solvers, and the marginal impact of the different reformulation techniques can be made.

Relative performance of the various solvers (base specifications). When considering the base specifications only, it can be observed that there is no a single solver winning on all problems: although OPL (the only commercial system considered) seems unbeatable on Word design, and to be able to solve larger instances of other problems (e.g., Golomb rulers and Magic squares) with respect to the other systems, ASP solvers prove to be competitive on several others, and in some cases they are much more efficient. Good examples are Ramsey, Car sequencing, Langford numbers, and All-interval series.

Among the ASP solvers, DLV seems to be more efficient than SMOBELS on the average; however, there are problems, like Still life and Langford numbers, where SMOBELS wins when compared to DLV.

Finally, the notable performance of CMOBELS deserve special attention: this solver is in fact almost always the best among the ASP ones (as it emerges also from related work [11]), and is in many cases better than OPL (in several other cases remaining competitive). This shows that SAT can provide a promising and effective technology for solving constraint problems, confirming the thesis in [5], where another SAT compiler, SPEC2SAT [6], is used to solve a number of combinatorial problems, specified in the modelling language NP-SPEC.

Impact of symmetry-breaking. From the experiments, it can be observed that symmetry-breaking may be beneficial on all solvers, although the complexity of the adopted symmetry-breaking constraints (cf. Figure 1) needs to be carefully chosen. As an example, DLV performs much better on the Ramsey problem with LI symmetry-breaking constraints, but such performance is not maintained when the more complex SB schema is adopted. A similar behavior can be observed on SMOBELS and CMOBELS.

As for Social golfer, Table 3 does not show significant performance improvements when symmetry-breaking is applied, with the ASP solvers (especially SMOBELS) being significantly slowed down when adopting the most complex schemas (LX and L2). However, it is worth noting that, on smaller negative (non-benchmark) instances, impressive speed-ups have been obtained for all systems, especially when using SA. As for LX, we also observe that it can be applied

Table 2: Experimental results for optimization problems. Entries denote the size of the largest instances solved by OPL, DLV, SMOBELS, and CMOBELS in 1 hour, using the base specification and their reformulations.

| Problem | Spec | OPL | DLV | SMODELS | CMODELS |
|--|--------------------|---------------------------------|---------------------------------|-----------------------------------|---------------------------------|
| Ramsey (nb. of nodes) | Base | 16 | 9 | 9 | 16 |
| | SA | 16 ⁻ | 9 ⁺ | 9 ⁺ | 16⁺ |
| | LI | 13 | 16 | 10 | 16 ⁻ |
| | SB | 16 ⁻ | 9 ⁺ | 8 | 13 |
| | Aux | 16 | 9 ⁺ | 9* | 16* |
| | Aux+SB | 16 ⁻ | 8 | — | — |
| | Aux+LI | 13 | 16 | — | — |
| Golomb rulers (nb. of marks) | Base | 10 | 9 | 6 | 6 |
| | SO | 10 ⁺ | 9 ⁻ | 6 | 6 ⁻ |
| | Aux | 11 | 9 | 8 | 9 |
| | GC(alldiff) | 11 | — | — | — |
| | Aux+SO | 11 | 9 ⁻ | 8 | 9 |
| | Aux+GC(alldiff) | 11 | — | — | — |
| | Aux+GC(alldiff)+SO | 12 | — | — | — |
| Water buckets (total time [sec]) (Perf. gain wrt Base spec.) | Base | 487.27 | 2056.85 | 27229.09 | 637.57 |
| | Aux | 233.23 (2.09x) | 323.26 (6.36x) | 257.88 (105.59x) | 39.83 (16.01x) |
| Max density Still life (board size) | Base | 8 | 7 | 8 | 8 |
| | SO | 8 | 7 | 8 ⁺ | 8 |
| | SB | 8 | 7 | 8 ⁻ | 7 |
| | Aux | 8 ⁻ | 6 | 8 ⁻ | 7 |
| | Aux+SO | 8 ⁻ | 7 | 8 ⁻ | 8 ⁻ |
| | Aux+SB | 8 ⁻ | 7 | 8 ⁻ | 7 |

The symbols and layout issues that denote some entries have the following meaning:

“+” (resp. “-”): although there is no variation in the size of the largest instance solved by the same solver with respect to the base specification, computation times are significantly lower (resp. higher);

Bold entries: denote the best specification for each solver, in terms of computation time to solve the largest instance;

Boxed entries: denote the best results, among all solvers, in terms of computation time to solve the largest instances.

Table 3: Experimental results for decision problems. The meaning of values in the entries is the same of those in Table 2.

| Problem | Spec | OPL | DLV | SMODELS | CMODELS |
|--|---------------------------|---|--------------------------------------|--------------------------------------|---|
| Social golfer (nb. of weeks) | Base | 5 | 6 | 6 | 6 |
| | SA | 5 | 6 ⁺ | 6 ⁺ | 6 |
| | LX | 5 | 6 | 1 | 6 |
| | L2 | 5 | 6 | 0 | 6 ⁻ |
| | Aux | 5 ⁺ | 6 | 6 ⁺ | 6 ⁺ |
| | GC(distr) | 3 | - | - | - |
| | Aux+SA | 5 ⁺ | 6 ⁺ | 6 ⁺ | 6⁺ |
| | Aux+GC(distr) | 6 | - | - | - |
| Car Sequencing (nb. of classes) (bench. 4/72, 6/76, 10/93) | Base | (10, 6, 12) | (13 , 9 , 12) | (13 , 9 , 12) | (13, 9, 15) |
| | SO | (10, 6, 12) | (13, 9, 12) | (13, 9, 12) | (13, 9, 15⁺) |
| | Aux | (10 ⁺ , 6 , 12) | (13, 9, 12) | (13, 9, 12) | (13⁺ , 9⁺ , 14) |
| | GC(distr) | (10 ⁺ , 6, 12) | (-, -, -) | (-, -, -) | (-, -, -) |
| | Aux+SO | (10 ⁺ , 6, 12) | (13, 9, 12) | (13, 9, 12) | (13 ⁺ , 9 ⁺ , 15 ⁻) |
| | Aux+GC(distr) | (10⁺ , 6, 12) | (-, -, -) | (-, -, -) | (-, -, -) |
| | Aux+SO+GC(distr) | (10 ⁺ , 6, 12) | (-, -, -) | (-, -, -) | (-, -, -) |
| Word Design DNA (nb. of words) | Base | 86 | 9 | 8 | 47 |
| | SO | 86 | 9 ⁻ | 13 | 45 |
| | SB | 56 | 5 | 5 | 18 |
| | LX | 87 | 4 | 14 | 43 |
| | Aux | 86 | 11 | 11 | 46 |
| | GC(distr) | 86 | - | - | - |
| | Aux+GC(distr) | 86 | - | - | - |
| | Aux+LX | 87 | 5 | 11 | 44 |
| | Aux+SO | 86 ⁻ | 9 ⁻ | 31 | 46 |
| Aux+SB | 56 | 5 | 11 | 27 | |
| Magic squares (square size) | Base | 4 | 3 | 2 | 2 |
| | LI | 4 ⁺ | 3 ⁺ | 2 ⁺ | 2 |
| | SA | 4 ⁻ | 3⁺ | 2 ⁺ | 2 |
| | SO | 4 ⁻ | 3 | 2 ⁺ | 2 |
| | Aux | 4⁺ | 3 ⁺ | 3 | 2 |
| | GC(alldiff) | 4 | - | - | - |
| | Aux+GC(alldiff) | 4 ⁺ | - | - | - |
| | Aux+GC(alldiff)+LI | 4 ⁺ | - | - | - |
| | Aux+LI | 4 ⁺ | 3 ⁺ | 3 | 2 |
| | Aux+SA | 4 ⁺ | 3⁺ | 3 | 2 |
| Langford number (nb. of values) (sat inst, unsat inst.) | Base | (12, 9) | (19, 10) | (24, 10) | (32, 10) |
| | SO | (12, 6) | (19, 10 ⁺) | (24, 10 ⁺) | (32 ⁻ , 10) |
| | GC(distr) | (12 ⁺ , 6) | - | - | - |
| | Aux | (16, 10) | (36, 10 ⁺) | (28 , 10 ⁺) | (99 , 10 ⁺) |
| | Aux+SO | (16, 10) | (36 , 10 ⁺) | (27, 10⁺) | (88, 10 ⁺) |
| | Aux+GC(alldiff) | (16, 10) | - | - | - |
| | Aux+GC(alldiff)+GC(distr) | (16 , 10) | - | - | - |
| Aux+GC(alldiff)+GC(distr)+SO | (16, 10) | - | - | - | |
| All-interval series (nb. of integers) | Base | 15 | 13 | 11 | 16 |
| | SO | 15 | 12 | 10 | 17 |
| | Aux | 15 ⁺ | 17 | 14 | 19 |
| | GC(alldiff) | 16 | - | - | - |
| | Aux+SO | 15 ⁻ | 13 ⁻ | 13 | 20 |
| | Aux+SO+GC(alldiff) | 6774 | - | - | - |

in two different ways, i.e., forcing either players’ schedulings or weekly groupings to be lexicographically ordered.²⁰ Values reported in Table 3 are obtained by lexicographically ordering weekly groupings: as a matter of fact, ordering players’ schedulings is even less efficient on SMOBELS, being comparable for the other solvers.

General rules for determining the right “amount” of symmetry breaking for any given solver on different problems are currently still unknown, but it seems that the simplest ones like SA or LI have to be preferred when using ASP solvers. On the other hand, from the experiments it results that OPL may benefit also from the addition of more complex schemas, like LX (cf., e.g., Word design). This is likely due to the more robust algorithms available in OPL for dealing with numbers and arithmetics (cf. also the forthcoming paragraph on this subject).

Figure 3 syntetically shows some of the lessons learned from our experiments regarding the amount of symmetry-breaking that should be worth applying with the various solvers. It has been obtained by considering the symmetry-breaking schemas that systematically led to better (or at least no worse) performance. Of course, exceptions may exist (cf., e.g. the good performance of LX on Word Design when solved with SMOBELS). However, these can be considered as a good starting-point to choose schemas to apply on a new problem.

Impact of global constraints. Experiments confirm that OPL may benefit from the use of global constraints. As an example, the base specification of the Golomb rulers problem encodes the constraint that forces the differences between pairs of marks to be all different by a set of binary inequalities. By replacing them with an `alldifferent` constraint, OPL was able to solve the instance with 11 marks in the time-limit, and time required to solve smaller instances significantly decreases. Also the Social golfer specification can be restated by using global constraints, in particular the `distribute` constraint. However, in this case our results show that OPL does not benefit from such a reformulation, in that it was not able to solve even the 4-weeks instance (solved in about 11 seconds with the base specification). Global constraints help OPL also on other problems, e.g., All-Interval series (`alldifferent`) and Car sequencing (`distribute`), even if, for the latter problem, the performance improvements don’t make it able to solve larger instances. Finally, Word design seems not to be affected by the introduction of `distribute`.

Impact of auxiliary predicates. As already mentioned, auxiliary predicates may in principle increase the size of the search space, but in several cases they result in better performance, especially when their introduction leads to great simplifications of complex constraints.

Actually, experiments show that, almost always, ASP solvers benefit from the use of auxiliary predicates (often not really needed by OPL, which allows to express more elaborate constraints), especially when they are defined relying on

²⁰None of the systems provide built-in constructs for expressing lexicographical orderings: hence, they have been modelled as ordinary constraints.

```

function solveMinimizationProblem(spec, instance) : solution or "unsat" {
  return findMin(spec, instance, lowerBound..upperBound);
  // Value for upperBound has been chosen as described in Section 3.2.1.
  // Value for lowerBound is usually 0.
}

function findMin(spec, instance, objValueRange) : solution or "unsat" {
  let sol = solveDecisionProblem(spec U "objValue in objValueRange", instance);
  if (sol == "unsat") return "unsat";

  let obj = objective value of "sol";
  let middle = objValueRange.min + floor((obj - objValueRange.min)/2);

  let solLeft = findMin(spec, instance, objValueRange.min..middle-1);
  if (solLeft != "unsat") return solLeft;
  else {
    let solRight = findMin(spec, instance, middle..obj-1);
    if (solRight != "unsat") return solRight;
    else return sol;
  }
}

```

Figure 2: Binary search algorithm used to solve minimization problems with CMODELS.

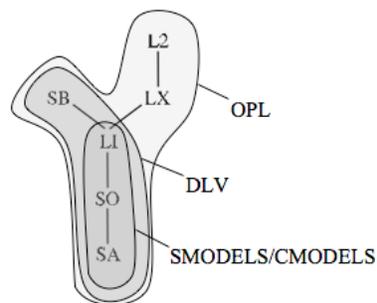


Figure 3: Symmetry-breaking schemas that systematically improve (or at least do not degrade) performance of the various solvers. They may be considered as the “best candidates” for a new problem with respect to the various solvers.

the minimal model semantics of ASP (hence, in rule heads). Good examples are Golomb rulers, where adding the auxiliary predicate `difference/2` leads to a strong improvement of all solvers except DLV, Water buckets, Langford numbers (where positive instances are solved much faster by all systems considered), All-interval, and the Social golfer problem, where SMODELS is able to solve the 6-weeks instance in 6 seconds when the auxiliary `meet/3` predicate is used, while solving the base specification requires 41 minutes (a similar behavior is observed with the other solvers).

Moreover, the addition of auxiliary predicates in many cases interestingly leads to smaller ground programs and to speed-ups of the grounding processes, because of the smart bottom-up techniques exploited by DLV and LPARSE. In particular, for ASP systems, such issues seem to be highly correlated: better performance of the Aux specifications are often accompanied by reductions of the respective ground programs (often of more than 50%), and with a speed-up of the grounding phase. It is worth noting that the latter issue may play a major role in the overall performance, since grounding may dominate the overall solving time (cf. also [5]). As an example, grounding times for instances of the Langford number problem greatly dropped down with both DLV and LPARSE (in particular, 1 minute was enough to ground the ‘36’ instance with DLV and the Aux specification, while the grounder did not terminate in 14 hours with the Base specification; as for LPARSE, it aborted while grounding the ‘99’ instance with the Base specification, but terminated in just 9 seconds with the Aux one). This behavior may be observed on several other problems (e.g., Magic squares, Social golfer, and All-interval series; for the last problem, grounding sped-up of more than 90% with both DLV and LPARSE, with similar reductions in size of the ground programs).

Also OPL often benefits from the addition of auxiliary predicates, but only after a simple search procedure that excludes branches on its variables has been added, cf. Section 3.2. The only exceptions are Max density Still life and Word Design DNA, for which no real improvements (and however no worse performance) have been observed.

It is worth noting that only in very few cases solvers suffer from the addition of auxiliary predicates (cf., e.g., results of the Maximum density still life problem, where, as for ASP solvers, we also observed no reduction –DLV– or even a strong growth –LPARSE– of the size of produced ground programs).

Synergic reformulations. Specifications obtained by combining, for each problem and solver, the most two efficient techniques, in many cases further boost performance, or at least do not affect it negatively. This is the case of, e.g., Golomb rulers, Social golfer, Car sequencing, Word design (OPL and SMODELS), Magic squares, Langford numbers, and All-interval series (CMODELS), that often proved to be able to deal with larger instances, or to be able to run significantly faster. Few exceptions do exist, e.g., Maximum density still life, where solving times were not significantly influenced. This gives evidence that combining “good” reformulations is in general a promising strategy to further boost

performance of all solvers.

Tables 2 and 3 also show some of the results obtained by other possible combinations, without considering any uniformity criteria. It can be observed that in few cases even better results could be achieved (cf., e.g., OPL on Golomb rulers or All-interval series and the specification with auxiliary predicates, global constraints and symmetry-breaking, or Social golfer and the specification with auxiliary predicates and global constraints), but in several others, only worse results were obtained. A deep understanding of the reasons of these further synergies will be an important topic for future research.

Impact of numbers and arithmetic constraints. As introduced at the end of Section 3.1, we evaluated the impact of numbers and arithmetics on the performance of the various solvers on two problems, Water buckets and Magic squares. In particular, we built new instances, equivalent to the original ones, but such to force the various solvers to deal with larger numbers, and measured the degradation of performance obtained by the most efficient specifications. In detail, we proceeded as follows:

- As for Water buckets, we computed the new instances by doubling both the buckets capacities and the water contents in the start and goal states;
- As for Magic squares, we changed the domain of values for each cell from the interval $[1, n]$ (n being the board size) to the set $\{i \cdot k \mid i \in [1, n]\}$ with k being 2 or 3.

We then compared the time needed by the different solvers to solve all instances (for what concerns Water buckets) and the largest instance reported in Table 3 (for what concerns Magic squares) with the most efficient specification (that with auxiliary predicates), with the time needed to deal with the instance obtained by performing the above modifications (to ensure fairness, no timeout has been enforced in this case).

Results are shown in Table 4. From there, it can be observed that, as expected, the presence of larger numbers and arithmetic constraints over them is a major obstacle for all solvers. However, from the results it follows that globally, OPL behaves much better than the ASP solvers on these issues, presumably because of the more powerful arc consistency techniques provided by this solver. Nonetheless, we must also pinpoint the good overall performance of CMODELS with respect to the other ASP solvers, which is likely to depend on the higher performance of current SAT algorithms already discussed above.

The price for declarativeness. As already stated in Section 2, in this paper we relied on the default behavior of the different solvers, avoiding the *ad-hoc* specification of any procedural aspects. In fact, we remind that the purpose of our experiments was not to solve the various problems in the most efficient ways as possible, but to understand what performance the different systems may offer when used in a transparent way (i.e., without requiring the user to

| Water buckets (Aux) | | | | |
|---------------------|----------|----------------------------|----------|----------|
| Instance | OPL | DLV | S MODELS | C MODELS |
| All | 233.23 | 323.26 | 257.88 | 39.83 |
| All (doubled) | 10383.04 | > 807614.87 ^(*) | 43205.64 | 5965.83 |
| Perf. degradation | 44.52x | > 3131.75x | 167.54x | 149.78x |

(*) We forced the termination of DLV runs on 4 of the doubled instances after 200,000 seconds. Hence, we are able to state only a lower bound of the performance degradation.

| Magic squares (Aux) | | | | |
|------------------------------|-------|-------|----------|----------|
| Instance | OPL | DLV | S MODELS | C MODELS |
| Largest solved (cf. Table 3) | 0.09 | 14.29 | 111.58 | 0.02 |
| Largest with $k = 2$ | 0.18 | 29.18 | 1727.81 | 0.11 |
| Perf. degradation | 2.00x | 2.04x | 15.48x | 5.50x |
| Largest with $k = 3$ | 0.17 | 41.14 | 6472.53 | 0.19 |
| Perf. degradation | 1.89x | 2.87x | 58.01x | 9.50x |

Table 4: Performance degradation due to the impact of numbers and arithmetic constraints for the various solvers (all times are in seconds, with no timeout enforced when solving modified instances).

take into account details about their underlying technologies), and which is the marginal impact of different high-level and strongly standardized reformulation techniques on solvers’ behavior.

It should be clear that a price must be paid when using systems like OPL in a fully declarative way, given that the standard default search procedure automatically added by the system is very general purpose, and does not take into account any structural aspect of the problem considered.

Nonetheless, in what follows we show that even in this case, performance obtained by making high-level (hence, in principle automatable) reformulation may greatly reduce this gap.

As an example, by combining three different techniques in the Golomb rulers problem, OPL was able to solve the “12” instance in less than one hour (the base specification was able to deal at most with the “10” instance). This seems to be a good result, when compared with the best ones reported, e.g., those obtained with the Koalog toolkit²¹, where a solution for instance “13” has been computed in 3.5 hours (cf. also the CSPLib problem description).

As another example, consider the Word design for DNA problem: our OPL model obtained by adding the LX symmetry-breaking schema to the base specification was able to solve the instance having size 87 in about 554 seconds. Ac-

²¹cf. http://www.koalog.com/php/jcs_golomb.php.

According to CSPLib, the instance having size 112 has been solved using Eclipse²² with many difficulties, by combining greedy and optimization approaches with the help of a linear programming tool and well chosen search heuristics. CSPLib also offers an OPL model for this problem, that provides a search procedure and explicitly lists all allowed strings, thus omitting some of the constraints (both these issues of course strongly affect declarativeness). However, this latter model was unable to solve largest instances than ours in the same time limit when run on our machine.

On the other hand, there are problems which have been solved more efficiently by ad-hoc built constraint programs: this is the case of, e.g., Magic squares and Maximum density Still life. As for the former, a good formulation is available for the Eclipse language (cf. the Eclipse web-site): the largest instance that such formulation is able to solve in one hour is that having size 5 (since it needs 2.5 hours to deal with the “6” instance on our machine). Our best OPL model (Aux) is able to solve the “4” instance in few hundreds of a second.

As for the latter problem instead, OPL was able to solve the instance having size 8 in about 120 seconds by applying SB symmetry-breaking. To the best of our knowledge, the largest instances solved by CP in reasonable time have size 9 (57 secs) and 10 (> 2 hours) [32], while by combining CP and Integer Programming and by performing several optimizations on the problem model, the instance of size 11 has been solved in 373 seconds [1] (instance “12” took almost 9 hours).²³

5 Conclusions

In this paper we reported results about an experimental investigation which aims at comparing the relative efficiency of a commercial backtracking-based and three academic ASP solvers when used in a purely declarative way, and the marginal impact that different and complementary reformulation techniques have on the different solving technologies.

In particular, we modelled 10 problems from the CSPLib into the languages used by the different solvers, in a way as systematic as possible, and applied various high-level reformulation techniques, also evaluating synergies among them.

Results show that there is not a single solver winning on all problems, with ASP being comparable to OPL for many of them. The good overall performance of the SAT-based ASP solver CMODELS clearly emerges from the experiments, thus confirming that the impressive advancements in SAT solvers should lead to definitively consider this technology as one the most promising approaches to solve combinatorial problems.

²²cf. <http://eclipse.crosscoreop.com>.

²³We reported also results beyond the one-hour time limit to take into account the obvious hardware differences. However, it is clear that this comparison can be nothing else than qualitative.

Furthermore, experiments clearly show that reformulating the specification almost always improves performance, and that high-level general reformulation schemes (that can hence be performed automatically by the system) can be worth. Experiments suggest several good modelling practices that may be followed when dealing with a new problem: structural symmetry-breaking may be beneficial, but a complexity threshold should not be exceeded for any given solver (cf. Figure 3); adoption of global constraints is usually worth when using OPL (especially *all-different*, while extra care must be used when exploiting *distribute*); addition of auxiliary predicates (especially for ASP solvers) is very welcome; finally, synergic combinations of the most promising techniques usually don't hurt. Notwithstanding this, an exact understanding of which reformulations lead to the best performance for a given problem and solver unsurprisingly remains a challenge.

Finally, we also performed an investigation about the impact of numbers and arithmetic constraints in problem specifications, and highlighted the better overall behavior of OPL with respect to ASP solvers in this context. However, also about this issue, the efficiency of the SAT approach emerges, with the worse performance of DLV and SMOELS with respect to CMOELS.

References

- [1] R. Bosch and M. Trick. Constraint programming and hybrid formulations for three life designs. In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2002)*, pages 77–91, Le Croisic, France, 2002.
- [2] M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. *Artificial Intelligence*, 170(8–9):779–801, 2006.
- [3] M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. *Applied Artificial Intelligence*, 21(4/5), 2007.
- [4] M. Cadoli, T. Mancini, D. Micaletto, and F. Patrizi. Evaluating ASP and commercial solvers on the CSPLib. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI 2006)*, pages 68–72, Riva del Garda, Trento, Italy, 2006. IOS Press, Amsterdam.
- [5] M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. In *Proceedings of the Sixteenth International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, volume 4203 of *Lecture Notes in Computer Science*, pages 540–549, Bari, Italy, 2006. Springer.
- [6] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.

- [7] E. Castillo, A. J. Conejo, P. Pedregal, R. Garca, and N. Alguacil. *Building and Solving Mathematical Programming Models in Engineering and Science*. John Wiley & Sons, 2001.
- [8] B. M. W. Cheng, K. M. F. Choi, J. H.-M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
- [9] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, Cambridge, MA, USA, 1996. Morgan Kaufmann, Los Altos.
- [10] A. K. Dewdney. The game life acquires some successors in three dimensions. *Scientific American*, 224(2):112–118, 1987.
- [11] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proceedings of the Twentyfirst International Conference on Logic Programming (ICLP 2005)*, volume 3668 of *Lecture Notes in Computer Science*, pages 67–82, Sitges, Spain, 2005. Springer.
- [12] T. Ellman. Abstraction via approximate symmetry. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 916–921, Chambéry, France, 1993. Morgan Kaufmann, Los Altos.
- [13] A. J. Fernández and P. M. Hill. A comparative study of eight constraint programming languages over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000.
- [14] R. A. Finkel, V. W. Marek, and M. Truszczynski. Constraint Lingo: towards high-level constraint programming. *Software — Practice and Experience*, 34(15):1481–1504, 2004.
- [15] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, page 462 ff., Ithaca, NY, USA, 2002. Springer.
- [16] R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
- [17] E. C. Freuder and D. Sabin. Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 191–196, Providence, RI, USA, 1997. AAAI Press/The MIT Press.

- [18] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.
- [19] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability Problem: Theory and Applications, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.
- [20] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
- [21] Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver enhanced to non-tight programs. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the Seventh International Conference on Logic for Programming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *Lecture Notes in Computer Science*, pages 346–350, Fort Lauderdale, FL, USA, 2004. Springer.
- [22] F. Lin and Z. Yuting. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.
- [23] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proceedings of the Sixth International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 165–181, Airth Castle, Scotland, UK, 2005. Springer.
- [24] T. Mancini and M. Cadoli. Exploiting functional dependencies in declarative problem specifications. *Artificial Intelligence*, 2007. To appear.
- [25] A. Neumaier, O. Shcherbina, W. Huyer, and T. Vinkó. A comparison of complete global optimization solvers. *Mathematical Programming*, 103(2):335–356, 2005.
- [26] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
- [27] N. Pelov, E. De Mot, and M. Denecker. Logic Programming approaches for representing and solving Constraint Satisfaction Problems: A comparison. In M. Parigot and A. Voronkov, editors, *Proceedings of the Seventh International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, volume 1955 of *Lecture Notes in Computer Science*, pages 225–239, Reunion Island, FR, 2000. Springer.
- [28] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In H. J. Komorowski and Z. W. Ras, editors, *Proceedings of the Seventh International Symposium on Methodologies for Intelligent Systems*

- (*ISMIS'93*), volume 689 of *Lecture Notes in Computer Science*, pages 350–361, Trondheim, Norway, 1993. Springer.
- [29] A. Ramani, F. A. Aloul, I. L. Markov, and K. A. Sakallak. Breaking instance-independent symmetries in exact graph coloring. In *Proceedings of Design Automation and Test Conference in Europe (DATE 2004)*, pages 324–331, Paris, France, 2004. IEEE Computer Society Press.
 - [30] J.-C. Régin. Global constraints and filtering algorithms. In M. Milano, editor, *Constraint and Integer Programming – Toward a Unified Methodology, Operations Research/Computer Science Interfaces*, volume 27, chapter 4. Kluwer Academic Publisher, 2003.
 - [31] O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.-H. Vu, and T.-V. Nguyen. Benchmarking global optimization and constraint satisfaction codes. In *Proceedings of the First International Workshop on Global Constraint Optimization and Constraint Satisfaction (COCOS 2002)*, volume 2861 of *Lecture Notes in Computer Science*, pages 211–222, Valbonne-Sophia Antipolis, France, 2003. Springer.
 - [32] B. M. Smith. A dual graph translation of a problem in ‘life’. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 402–414, Ithaca, NY, USA, 2002. Springer.
 - [33] B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, pages 182–187, Austin, TX, USA, 2000. AAAI Press/The MIT Press.
 - [34] G. Smolka. The Oz programming model. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 1995.
 - [35] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1988.
 - [36] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
 - [37] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On benchmarking constraint logic programming platforms. response to [13]. *Constraints*, 9(1):5–34, 2004.
 - [38] T. Walsh. Permutation problems and channelling constraints. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the Eighth International Conference on Logic for Programming and Automated Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Computer Science*, pages 377–391, Havana, Cuba, 2001. Springer.