

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`

March 9, 2007

Abstract

This paper¹ deals with four solvers for combinatorial problems: the commercial state-of-the-art solver Ilog OPL, 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

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

1
2
3
4
5
6
7
8
9 Programming solvers², has been challenged by the use of libraries for Con-
10 straint Programming (CP), such as Ilog SOLVER³, interfaced through classical
11 programming languages, e.g. C++ or Prolog. At the same time, the need for
12 a higher level of abstraction and declarativeness led to the design and develop-
13 ment of general purpose languages for constraint modelling and programming
14 –e.g. OPL [31], XPRESS^{MP4} or GAMS [7]– and languages based on specific solvers,
15 such as AMPL [15], DLV [18], SMOBELS [22], CMOBELS [19] or ASSAT [20].

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

21 On the other hand, with these languages, it is now the *quality* of the problem
22 model that plays a fundamental role in solvers' efficiency: it is in fact very well-
23 known that the most simple, intuitive, and straightforward formulations for a
24 given problem rarely are the most efficient ones, and several techniques have
25 been proposed in the literature in order to reformulate the original model into
26 equivalent ones that are more efficiently evaluable by the solver at hand. Such
27 techniques include symmetry-breaking (cf., e.g., [9, 24, 14, 21]), abstraction
28 of constraints (cf., e.g., [17, 12, 16, 2]), addition of implied constraints and of
29 auxiliary variables (cf., e.g., [28, 33]), and use of global constraints (cf., e.g., [26]).

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

41 Such evolution is actually not new: a similar process has been experienced
42 during the past decades in the development of languages and systems for query-
43 ing databases: without the existence of effective tools for query preprocessing
44 and optimization (cf., e.g., [30]), the relational model and modern query lan-
45 guages like SQL would improbably spread through real applications.

46
47
48
49 With such a long-term goal in mind, in previous work [2, 21, 1, 3] we showed how
50 problem models can be regarded as logical formulae, how several of the afore-
51 mentioned techniques for model reformulation may be generalized and highly
52 standardized, and how automated tools can be practically used in order to check
53 their applicability for the problem at hand. This led to the possibility, by the

54
55 ²e.g. Ilog CPLEX, cf. <http://www.ilog.com/products/cplex>.

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

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

1
2
3
4
5
6
7
8
9 solving engine, to effectively perform specification-level analysis of the user’s
10 model, and automatize several reformulation tasks.

11
12 In this paper we consider four well-known systems for constraint solving, and
13 perform an extensive experimentation in order to answer the following main
14 questions:

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

18
19 The last question poses, in turn, two more issues:

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

24
25 The systems we consider are one commercial state-of-the-art solver, i.e., Ilog OPL
26 and three ASP solvers, namely DLV, SMOBELS, and CMOBELS. The latter two
27 have the interesting property of sharing the specification language with several
28 other solvers through the common parser LPARSE⁵. Importantly, while DLV and
29 SMOBELS use proprietary solving engines, CMOBELS compiles ASP specifications,
30 when given together with an instance, into an instance of the Propositional
31 Satisfiability problem (SAT), for which many efficient solvers are today available.

32 At the higher level, such systems exhibit interesting differences, including
33 availability (OPL and ASP solvers are, respectively, payware and freeware sys-
34 tems, the latter often being open source), algorithm used by the solver (resp.
35 backtracking- and fixpoint-based), expressiveness of the modelling language
36 (e.g., availability of arrays of finite domain variables vs. boolean matrices), com-
37 pactness of constraint representation (e.g., availability of global constraints). In
38 order to answer the questions stated above, we present a reproducible and ex-
39 tensible experimental methodology. In particular, we consider a third-party
40 problem library, i.e. the CSPLib⁶, and uniform rules for modelling and instance
41 selection. As for the reformulation techniques subject of our experiments, we
42 consider *symmetry breaking*, the use of *global constraints*, and that of *auxiliary*
43 *variables and predicates*. Moreover, we try to assess how negatively *numbers*
44 *and arithmetics* influence performance.

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

50
51
52
53
54
55 ⁵cf. <http://www.tcs.hut.fi/Software/smodels>.

56 ⁶cf. <http://www.csplib.org>.

1
2
3
4
5
6
7
8
9 Comparison among different solvers for CP has already been addressed in the
10 literature: in particular, we recall [13] and [32] where SOLVER is compared to
11 other CP languages such as, e.g., OZ [29], CLAIRE⁷, and various Prolog-based
12 systems. Moreover, some benchmark suites have been proposed, cf., e.g., the
13 COCONUT one [27]. Also on the ASP side, which has been the subject of much
14 research in the recent years, benchmark suites have been built in order to facil-
15 itate the task of evaluating improvements of their latest implementations, the
16 most well-known being ASPARAGUS⁸ and ASPLib⁹. However, less research has
17 been done in comparing solvers based on different formalisms and technologies,
18 and in evaluating the relative impact of different features and modelling tech-
19 niques. In particular, very few papers compare ASP solvers to state-of-the-art
20 systems for CP. To this end, we cite [11], where two ASP solvers are compared
21 to a CLP(FD) Prolog library on six problems: Graph coloring, Hamiltonian
22 path, Protein folding, Schur numbers, Blocks world, and Knapsack, and [23],
23 where ASP and Abductive Logic Programming systems, as well as a first-order
24 finite model finder, are compared in terms of modelling languages and relative
25 performances on three problems: Graph coloring, N-queens, and a scheduling
26 problem.
27
28

29
30 **Outline.** The outline of the paper is as follows: in Section 2 we present and
31 discuss the adopted experimental methodology, while in Section 3 we describe
32 the experimental framework, i.e., the selected problems, their various formula-
33 tions, and how we chose their instances. Then, in Section 4 we present and
34 analyze the results of our experiments. Finally, Section 5 concludes the paper.
35
36

37 2 Methodology

38

39 In this section we present the methodology adopted in order to achieve the
40 goals mentioned in Section 1. For each problem, we define a number of different
41 formulations: a *base specification*, obtained by a straightforward and intuitive
42 “translation” of the CSPLib problem description into the target language, and
43 several *reformulated* ones, obtained by using different techniques proposed in the
44 literature: (i) symmetry-breaking, (ii) addition of global constraints and (iii)
45 addition of auxiliary predicates. Moreover, in order to establish whether merging
46 different reformulations, which are proven to improve performance when used
47 alone, speeds-up even more the computation, we considered additional spec-
48 ifications for the same problems, obtained by *combining* the aforementioned
49 techniques, and exploring the existence of synergies among them. Finally, an
50 evaluation of the impact of numbers and arithmetic constraints in all the solvers
51 involved in the experimentation has been performed on two problems.
52

53 ⁷cf. <http://claire3.free.fr>.

54 ⁸cf. <http://asparagus.cs.uni-potsdam.de>.

55 ⁹cf. <http://dit.unitn.it/~wasp>.

56
57
58
59
60
61
62
63
64
65

1
2
3
4
5
6
7
8
9 The goals declared in Section 1 drive us to follow a purely declarative ap-
10 proach during modelling: hence, all the specifications have been solved without
11 performing any kind of tuning of the search parameters and without specify-
12 ing any procedural aspects (like, e.g., *ad hoc* search procedures in OPL). Thus,
13 the experimentation relies on the *default behavior* of the different solvers (i.e.,
14 grounding techniques, branching heuristics, etc.).

15 In fact, we remind that our purpose is not to solve a particular problem
16 in the most efficient way as possible, but to understand what performance the
17 various systems may offer to the modeller in a transparent way, i.e., without
18 requiring him to take into account any details about the underlying technologies,
19 and which is the effectiveness and the marginal impact of various high-level and
20 strongly standardized reformulation techniques on solvers' performance.

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

27 Due to the high number of problems and instances solved, the necessity of
28 having a synthetic measure of the various solvers performance arises. Hence, for
29 each problem we fix all the input parameters but one (such choices are discussed
30 in Section 3.1), and in our results we report, for each problem and solver, the
31 *largest instance* (denoted by the value given to the selected parameter) that is
32 solvable in a given time limit (one hour) by each reformulation.¹⁰
33
34

36 3 The experimental framework

38 3.1 Problems selection

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

44 For our experiments we chose 10 problems from CSPLib, that cover all the
45 7 areas of the collection. Since many problems in CSPLib are described only in
46 natural language, without any formal characterization, this work also provides,
47 as a side-effect, their formal specifications in the modelling languages adopted
48 by several solvers.

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

52 ¹⁰As discussed in Section 4, in order to neutralize some noise in the solvers' behavior, the
53 definition of *largest solvable instance* has to be slightly emended.
54
55
56
57
58

1
2
3
4
5
6
7
8
9 **Ramsey problem** (#017: Bin pack. and partitioning, Comb. math.). This
10 problem amounts to color the edges of a complete graph with n nodes using
11 the minimum number of colors, in order to avoid monochromatic triangles.
12 Instances of the problem are denoted by the number of graph nodes, n .
13

14
15 **Social golfer** (#010: Sched., Bin pack. and part., Games & puzzles). In a golf
16 club there are 32 *social* golfers who play once a week in 8 groups of 4. The
17 problem amounts to find a schedule for as many as possible weeks, such that
18 no two golfers play in the same group more than once. Here we consider a
19 decisional version of the problem: given a positive integer w , the goal is to find
20 a schedule for w weeks. Problem instances are thus denoted by the number of
21 weeks w .
22

23
24 **Golomb rulers** (#006: Frequency assign., Comb. math.). Given a positive in-
25 teger m , this problem amounts to put m marks on a ruler, in such a way that the
26 $m(m-1)/2$ distances among them are all different. The objective is to find the
27 length of the shortest ruler that admits the positioning of m marks. Instances
28 are denoted by values given to m . **Car sequencing** (#001: Scheduling). A
29

30 number of cars are to be produced; they are not identical, because different
31 options are available as variants on the basic model. The assembly line has
32 different stations which install the various options (air-conditioning, sun-roof,
33 etc.). Such stations have been designed to handle at most a certain percentage
34 of the cars passing along the assembly line. Furthermore, the cars requiring a
35 certain option must not be bunched together, otherwise the station will not be
36 able to cope with them. Consequently, the cars must be arranged in a sequence
37 so that the capacity of each station is never exceeded.
38

39 Selection of instances for this problem differs from the previous cases, since
40 they cannot be directly encoded by a single parameter. We considered some
41 benchmarks suggested in the CSPLib, namely “4/72”, “6/76” and “10/93”.
42 However, since they were too hard for our solvers, we proceeded as follows, in
43 order to generate a new set of (smaller) instances: from any original benchmark
44 (with n classes), we generated a set of instances by reducing the number of
45 classes to all possible smaller values, and consequently resizing station capacities
46 in order to avoid an undesirable overconstraining that would make instances
47 unfeasible. Thus, instances derived from the same benchmark could be ordered
48 according to the value for the (reduced) number of classes, which could then be
49 regarded as a measure for their size.
50

51
52 **Water buckets** (#018: Design, conf. and diagnosis, Bin pack. and part., Games
53 & puzzles). We consider a generalization of the CSPLib specification, which is
54 as follows: Given an 8 pint bucket of water, and two empty buckets which can
55 contain 5 and 3 pints respectively, the problem requires to divide the water
56 into two by pouring water between buckets (that is, to end up with 4 pints in
57
58

1
2
3
4
5
6
7
8
9 the 8 pint bucket, and 4 pints in the 5 pint bucket) in the smallest number of
10 transfers.

11 The generalization consists in making the specification parametric with re-
12 spect to the start and goal configurations, which are now inputs to the problem.
13 We also designed 11 instances (each one denoted by a start and a goal configura-
14 tion) from that considered in the original problem description, which have been
15 proved to be non-trivial by preliminary experiments. Since such instances could
16 not be denoted by a single parameter, solvers' performance for this problem and
17 the different specifications have been compared by considering the overall time
18 needed to solve them.
19

20
21 **Maximum density still life** (#032: Games & puzzles). This problem arises
22 from the Game of life, invented by John Horton Conway in the 1960s (cf.,
23 e.g., [10] for a detailed description). Life is played on a squared board, considered
24 to extend to infinity in all directions. Each square of the board is a cell, which
25 at any time during the game is either alive or dead. A cell has eight neighbours.
26 The configuration of live and dead cells at time t leads to a new configuration
27 at time $t + 1$ according to the rules of the game:
28

- 29 • If a cell has exactly three living neighbours at time t , it is alive at time
30 $t + 1$;
- 31
- 32 • If a cell has exactly two living neighbours at time t , it is in the same state
33 at time $t + 1$ as it was at time t ;
- 34
- 35 • Otherwise, the cell is dead at time $t + 1$.

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

40 Given a positive integer n , the problem amounts to find the densest possible
41 still-life pattern, i.e. the pattern with the largest number of live cells, that can
42 be fitted into an $n \times n$ section of the board, with all the rest of the board dead.
43 Instances are encoded by n , the size of the board.
44

45
46 **Word design for DNA computing on surfaces** (#033: Bioinformatics).
47 This problem amounts to find as large a set S of strings (words) of length 8 over
48 the alphabet $W = \{A, C, G, T\}$ with the following properties:
49

- 50 • Each word in S has 4 symbols from $\{C, G\}$;
- 51
- 52 • Each pair of distinct words in S differ in at least 4 positions; and
- 53
- 54 • Each pair of words x and y in S (where x and y may be identical) are
55 such that $R(x)$ and $C(y)$ differ in at least 4 positions, where:
56 – $R(x)$ is the reverse of the input string x ;
- 57

1
2
3
4
5
6
7
8
9 – $C(y)$ is the Watson-Crick complement of y , i.e. the word obtained by
10 y by replacing each A by a T and vice versa and each C by a G and
11 vice versa.
12

13 Here we consider a decisional version of the problem: given a positive integer c ,
14 the goal is to find, if possible, a set of words S with the above properties, and
15 such that $|S| = c$. Instances are encoded by c .
16

17
18 **Magic squares** (#019: Comb. math., Games & puzzles). An order n magic
19 square is a n by n matrix containing the numbers 1 to n^2 , with each row, column
20 and main diagonal equal the same sum. Given a positive integer n , the problem
21 amounts to find a n order magic square. Instances are encoded by n , the matrix
22 order.
23

24
25 **Langford numbers** (#024: Comb. math., Games & puzzles). This is a gener-
26 alization of the specification given in the CSPLib (which fixes the forthcoming
27 value n to 4). Given two sets of the numbers from 1 to n , the problem amounts
28 to arrange the $2n$ numbers in the two sets into a single sequence in which the
29 two 1's appear one number apart, the two 2's appear two numbers apart, the
30 two 3's appear three numbers apart, ..., and the two n 's appear n numbers
31 apart. Instances are encoded by n .
32

33
34 **All-interval series** (#007: Frequency assign., Comb. math.). Given the twelve
35 standard pitch-classes (c , $c\#$, d , ...), represented by numbers 0,1,...,11, this
36 problem amounts to find a series in which each pitch-class occurs exactly once
37 and in which the musical intervals between neighboring notes cover the full
38 set of intervals from the minor second (1 semitone) to the major seventh (11
39 semitones). That is, for each of the intervals, there is a pair of neighboring
40 pitch-classes in the series, between which this interval appears.
41

42 We consider a generalization of this problem in which the set of numbers is
43 the range from 0 to $n - 1$, for any given positive n . In particular, given such n ,
44 the problem amounts to find a vector $s = (s_1, \dots, s_n)$ that is a permutation of
45 $\{0, 1, \dots, n - 1\}$ and such that the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n -$
46 $s_{n-1}|)$ is a permutation of $\{1, 2, \dots, n - 1\}$. Instances are encoded by the integer
47 n .
48

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

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

56 Actually, two more problems, namely Social golfer and Word design, are
57 presented as optimization problems in the CSPLib. However, since none of our
58

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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	Benchmark instances
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.

solvers were able to achieve their optimal solutions,¹¹ in our experiments we considered their decisional versions.

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

¹¹We observe that, as for Social golfer, the longest schedule is still unknown in the literature, being of 9 or 10 weeks.

1
2
3
4
5
6
7
8
9 much their performance degrade, we are able to evaluate the robustness of the
10 various technologies with respect to such issues.

11 12 **3.2 Models selection**

13
14 As claimed in Section 2, in order to build an extensible experimental framework,
15 we followed the approach of being as uniform and systematic as possible during
16 the modelling phase, by requiring the specifications of the various solvers to
17 be similar to each other. Hence, even if not always identical because of the
18 intrinsic differences among the languages that could not be overcome, all of the
19 models share the same ideas behind the search space and constraints definitions,
20 independently of the language. Below, we discuss the general criteria followed
21 during the modelling phase, and the different formulations considered for each
22 problem. Encodings of all problems for all solvers are available in the web-
23 appendix of this paper.¹² We refer the reader to it for a detailed understanding
24 of the different problems' formulations described in what follows.

25 26 **3.2.1 General modelling criteria**

27
28 The first obvious difference between OPL and the ASP solvers concerns the search
29 space declaration. The former relies on the notion of *function* from a finite do-
30 main to a finite codomain, while the latter ones have just *relations*, which must
31 be restricted to functions through specific constraints. Domains of relations can
32 be implicit in DLV, since the system infers them by contextual information. For
33 each language, we used the most natural declaration construct, i.e, functions
34 for OPL, and untyped relations for DLV. Secondly, since the domain itself can
35 play a major role in efficiency, sometimes it has been inferred through some *a*
36 *posteriori* consideration. This is especially the case for domains of objective val-
37 ues in optimization problems. As an example, in Golomb rulers the maximum
38 position for marks (hence the rule length to be minimized) is upper-bounded
39 by 2^m (m being the number of marks), but choosing such a large number can
40 advantage OPL, which has powerful arc-consistency algorithms for domain re-
41 duction. As a consequence, we used the upper bound $3L/2$ for all solvers, L
42 being the maximum mark value for the optimum of the specific instance. Such
43 practice has been carried out systematically for each minimization problem (3
44 out of 4), by fixing the upper bound for the value of the objective function to
45 $3L/2$, with L being the optimum (computed by preliminary experiments). As
46 for the unique maximization problem considered (Maximum density still life),
47 as discussed in Section 4 (cf. also the problem description given in Section 3.1),
48 this problem does not apply.

51 ¹²cf. <http://www.dis.uniroma1.it/~tmancini/index.php?currItem=research.>
52 [publications.webappendices.csplib2x.](http://www.dis.uniroma1.it/~tmancini/index.php?currItem=research.)

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.

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:

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

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, 24, 14]), 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 [21]. 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):

- 1
 - 2
 - 3
 - 4
 - 5
 - 6
 - 7
 - 8
 - 9
 - 10
 - 11
 - 12
 - 13
 - 14
 - 15
 - 16
 - 17
 - 18
 - 19
 - 20
 - 21
 - 22
 - 23
 - 24
 - 25
 - 26
 - 27
 - 28
 - 29
 - 30
 - 31
 - 32
 - 33
 - 34
 - 35
 - 36
 - 37
 - 38
 - 39
 - 40
 - 41
 - 42
 - 43
 - 44
 - 45
 - 46
 - 47
 - 48
 - 49
 - 50
 - 51
 - 52
 - 53
 - 54
 - 55
 - 56
 - 57
 - 58
 - 59
 - 60
 - 61
 - 62
 - 63
 - 64
 - 65
- **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., [14]). 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 $s_1 < s_2$

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

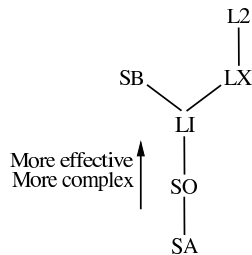


Figure 1: Classification among symmetry-breaking schemas as a partial ordered set. Higher schemas are more effective, but usually require more complex constraints.

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 [25] 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 global constraints, hence no comparison can be made on this issue.

1
2
3
4
5
6
7
8
9 **3.2.5 Reformulation by adding auxiliary predicates**

10 A predicate in the search space is called *auxiliary* if its extensions functionally
11 depend on those of the other ones. The use of auxiliary guessed predicates is
12 very common in declarative programming, especially when the user needs to
13 store partial results, to maintain intermediate states, or wants to make the so-
14 called *redundant modelling* (cf., e.g., [8]), by maintaining multiple views of the
15 search space, synchronized by channelling constraints.
16

17 As an example, consider the specification of Social golfer, where the search
18 space is a total function assigning a group to each player/week pair. We could
19 add to the search space an auxiliary guessed predicate $meet(\cdot, \cdot, \cdot)$ and suitable
20 channelling constraints that, for each pair of players p_1, p_2 ($p_1 \neq p_2$), and each
21 week w , force the triple $\langle p_1, p_2, w \rangle$ to belong to $meet$ iff players p_1 and p_2
22 are expected to play in the same group on week w , according to the guessed
23 scheduling.
24

25 It can be observed that any extension of the guessed scheduling uniquely
26 defines an extension for the $meet$ predicate (this is called a *functional dependence*
27 among guessed predicates —cf., e.g., [1]). The main advantage of using such
28 predicate is in the simplification of some of the constraints: as an example, the
29 *meet-at-most-once* constraint can be much more compactly expressed in terms
30 of the $meet$ predicate.
31

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

36 We consider equivalent specifications, obtained by using auxiliary predicates,
37 for all of the selected problems. However, the bottom-up evaluation algorithms
38 of DLV, SMOBELS, and CMOBELS may significantly advantage ASP solvers over
39 OPL on such specifications, since auxiliary predicates are usually defined in rule-
40 heads. To this end, when adding auxiliary predicates to OPL specifications, we
41 also added simple, general, and high-level modelled search procedures instruct-
42 ing the labelling algorithm to delay branches on auxiliary variables, while main-
43 taining the default behavior on the others, as explained in previous work [1].
44

45 **3.2.6 Reformulation by combining different techniques**

46 In many cases, more than one single reformulation strategy improves perfor-
47 mance on a given problem. Hence, the question arises whether synergies exist
48 among them, and what techniques are likely to work well together, for each
49 solver.
50

51 To this end, for each problem we consider some additional formulations: the
52 first one has been obtained, according to the aforementioned uniformity crite-
53 ria, by merging the two reformulations (among symmetry-breaking, addition
54 of global constraints and of auxiliary predicates) that, for each solver, resulted
55 to be the most efficient. Finally, in order to understand whether there exist
56 better, undiscovered synergies, we relaxed the uniformity hypothesis, and con-
57
58
59
60
61
62
63
64
65

sidered 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 show 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 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.
```

1
2
3
4
5
6
7
8
9 **Reformulation by symmetry-breaking: Selective ordering**

10 OPL
11
12 // ...Base specification plus:
13 ((ruler[2] - ruler[1]) < (ruler[n_marks] - ruler[n_marks - 1])); // S0
14

15
16 DLV
17
18 \\\% ...Base specification plus:
19 :- ruler(1,Y), ruler(2,Y1), ruler(M,Y3), ruler(N,Y2), \\\% S0
20 Y1=D1+Y, Y3=D2+Y2, D2<D1, M=N+1, n_marks(M).
21

22 SMODELS and CMODELS (LPARSE)
23
24 \\\% ...Base specification plus:
25 :- ruler(1,Y), ruler(2,Y1), ruler(n_marks-1,Y2), \\\% S0
26 ruler(n_marks,Y3), lt(Y1-Y,Y3-Y2).
27

28
29
30 **Reformulation by exploiting global constraints: all-different**

31 OPL
32
33 // ...Base specification, with c3 replaced by:
34 alldifferent(
35 all (i,j in 1..n_marks: i <> j) (ruler[j] - ruler[i])); // c3 (GC)
36
37

38 DLV, SMODELS, and CMODELS: not applicable.
39
40

41 **Reformulation by adding auxiliary predicates**

42 OPL
43
44 int+ n_marks = ...;
45 int+ maxval = ...;
46 int+ numOfDifferences = (n_marks*(n_marks-1)/2);
47 range diffValues 1..maxval;
48
49 var int+ ruler[1..n_marks] in 0..maxval; // search space decl.
50 var diffValues distance[1..numOfDifferences]; // aux. pred. decl.
51
52 minimize ruler[n_marks] // obj. func.
53 subject to {
54 ruler[1] = 0; // c1
55 forall (i in 1..n_marks-1) { // c2
56 ruler[i] < ruler[i+1];
57
58

```

1
2
3
4
5
6
7
8
9
10     };
11     forall(m1,m2 in 1..n_marks: m2 > m1){           // chann. constr.
12         distance[((n_marks*(n_marks-1)/2) -
13             ((n_marks-m1+1)*(n_marks-m1)/2) + (m2-m1))] =
14             ruler[m2] - ruler[m1]
15     };
16     forall(i,j in 1..numOfDifferences : i > j) {     // c3 (Aux)
17         distance[i] <> distance[j];
18     };
19
20 // Delay branches on aux. predicate
21 search{
22     generate(ruler);
23     generate(distance);
24 };
25
26
27 DLV
28
29 ruler(P,M) v fail(P,M) :- marks(M), positions(P). \\\% search space decl.
30 :- positions(P), not #count{ M : ruler(P,M)}=1.
31
32 ruler(1,0).                                           \\\% c1
33 :- ruler(X,Y), ruler(X1,Y1), X1>X, Y>=Y1.          \\\% c2
34 diff(X1,X2,D) :- ruler(X1,Y1), ruler(X2,Y2),        \\\% aux. pred. decl. and chann. constr.
35                 X2>X1, Y2=D+Y1.
36 :- #count{X1,X2: diff(X1,X2,D)}>1, marks(D), D>0. \\\% c3 (Aux)
37 :- ruler(P,M), n_marks(P). [M:1]                    \\\% obj. func.
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```

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:

```

1
2
3
4
5
6
7
8
9
10 int+ n_marks = ...;
11 int+ maxval = ...;
12 int+ numOfDifferences = (n_marks*(n_marks-1)/2);
13 range diffValues 1..maxval;
14
15 var int+ ruler[1..n_marks] in 0..maxval;           // search space decl.
16 var diffValues distance[1..numOfDifferences];      // aux. pred. decl.
17
18 minimize ruler[n_marks]                           // obj. func.
19 subject to {
20     ruler[1] = 0;                                  // c1
21     forall (i in 1..n_marks-1) {                  // c2
22         ruler[i] < ruler[i+1];
23     };
24     forall(m1,m2 in 1..n_marks: m2 > m1){         // chann. constr.
25         distance[((n_marks*(n_marks-1)/2) -
26             ((n_marks-m1+1)*(n_marks-m1)/2) + (m2-m1))] =
27             ruler[m2] - ruler[m1];
28     };
29     alldifferent(distance);                        // c3 (Aux+GC)
30     ((ruler[2] - ruler[1]) <                      // S0
31         (ruler[n_marks] - ruler[n_marks - 1]));
32 };
33
34 // Delay branches on aux. predicate
35 search{
36     generate(ruler);
37     generate(distance);
38 };

```

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 SOLVER v. 5.3, invoked through OPLSTUDIO 3.61,
- DLV v. 2005-02-23,
- SMOBELS v. 2.28, by using LPARSE 1.0.17 for grounding,
- CMOBELS 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.

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

1
2
3
4
5
6
7
8
9 of instances, which selection has been performed as discussed in Section 3.1. All
10 runs had a timeout of 1 hour.

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

15 However, during our experiments, we experienced that, for a given speci-
16 fication, sometimes solvers were unable to solve instance denoted by value k
17 of the parameter in the time-limit, but were able to solve the one denoted by
18 $k + 1$. This “noisy” (or “non-monotone”) behavior is very well-known in the
19 literature. To ensure fairness, and to avoid the effects of such a noisy behavior,
20 in the results *we intend for largest instance* the one denoted by the largest value
21 k of the parameter that satisfies the following three conditions: (i) It is solvable
22 in the time-limit by the given solver with the given specification; (ii) Neither
23 instance $k + 1$ nor instance $k + 2$ are solvable in the time-limit by the same
24 solver with the same specification; (iii) There not exist instances k' and $k' + 1$
25 ($k' \leq k - 2$) which are not solvable in the time-limit by the same solver with
26 the same specification.
27

28
29 We also remind that one solver, namely CMODELS, cannot natively deal with
30 optimization problems. Nonetheless, CMODELS has been run also to solve the
31 instances of the 4 optimization problems: its entries in Table 2 have been ob-
32 tained by iteratively invoking the solver several times, by adding each time new
33 constraints to the specifications that limit the values of the objective function
34 evaluated on the solutions found, by following the approach of performing a
35 binary search of the optimum objective value. Of course, such results (denot-
36 ing the largest instances for which the overall time needed to find their opti-
37 mum solutions was below the time-limit) represent lower bounds of the solver’s
38 performance. Nonetheless they show that, even in this case, CMODELS is com-
39 petitive with respect to the other solvers, thus confirming that SAT should be
40 undoubtedly regarded as one of the most promising and effective technology for
41 constraint problems to date (such evidence has also been discussed in related
42 work [5]). Figure 2 shows the algorithm used to solve minimization problems
43 (3 out of 4) with CMODELS. As for the unique maximization problem, i.e.,
44 Maximum density still life, we observe that a dual approach could be straight-
45 forwardly used, since maximizing the number of live cells does not increase the
46 size of the search space (the total number of variables remains unchanged).
47

48
49 As discussed in Section 3.1, the size of the instances for few problems could not
50 be easily represented by a single parameter. In such cases, in order to give a
51 synthetic yet meaningful measure of the performance of the various solvers, we
52 proceeded as follows: as for Car sequencing, we report, for each set of instances
53 generated from CSPLib benchmarks “4/72”, “6/76” and “10/93” (cf. the dis-
54 cussion about instance selection for this problems in Section 3.1), the largest
55 one solved, i.e., the one with the largest number of classes, and, as for Water
56
57
58

```

1
2
3
4
5
6
7
8
9 function solveMinimizationProblem(spec, instance) : solution or "unsat" {
10     return findMin(spec, instance, lowerBound..upperBound);
11     // Value for upperBound has been chosen as described in Section 3.2.1.
12     // Value for lowerBound is usually 0.
13 }
14
15 function findMin(spec, instance, objValueRange) : solution or "unsat" {
16     let sol = solveDecisionProblem(spec U "objValue in objValueRange", instance);
17     if (sol == "unsat") return "unsat";
18
19     let obj = objective value of "sol";
20     let middle = objValueRange.min + floor((obj - objValueRange.min)/2);
21
22     let solLeft = findMin(spec, instance, objValueRange.min..middle-1);
23     if (solLeft != "unsat") return solLeft;
24     else {
25         let solRight = findMin(spec, instance, middle..obj-1);
26         if (solRight != "unsat") return solRight;
27         else return sol;
28     }
29 }

```

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

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.

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	-	-
Golomb rulers (nb. of marks)	Aux+LI	13	16	-	-
	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])	Base	487.27	2056.85	27229.09	637.57
	Aux	233.23	323.26	257.88	39.83
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 use of auxiliary predicates seems to be unavoidable when modelling the Ramsey problem in SMOBELS and CMOBELS. For this reason, the Base and Aux specifications coincide.

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	S MODELS	C MODELS
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	-	-	-

1
2
3
4
5
6
7
8
9 **Relative performance of the various solvers (base specifications).** When
10 considering the base specifications only, it can be observed that there is no a single
11 solver winning on all problems: although OPL (the only commercial system
12 considered) seems unbeatable on Word design, and to be able to solve larger instances
13 of other problems (e.g., Golomb rulers and Magic squares) with respect
14 to the other systems, ASP solvers prove to be competitive on several others,
15 and in some cases they are much more efficient. Good examples are Ramsey,
16 Car sequencing, Langford numbers, and All-interval series.

17 Among the ASP solvers, DLV seems to be more efficient than SMOBELS on the
18 average (as it emerges also from related work [11]); however, there are problems,
19 like Still life and Langford numbers, where SMOBELS wins when compared to
20 DLV. Finally, the notable performance of CMOBELS deserve special attention:
21 this solver is in fact almost always the best among the ASP ones, and is in many
22 cases better than OPL (in several other cases remaining competitive). This shows
23 that SAT technology can provide a promising and effective technology for solving
24 constraint problems, confirming the thesis in [5], where another SAT compiler,
25 SPEC2SAT [6], is used to solve a number of combinatorial problems, specified
26 in the modelling language NP-SPEC.
27

28
29 **Impact of symmetry-breaking.** From the experiments, it can be observed
30 that symmetry-breaking may be beneficial on all solvers, although the complex-
31 ity of the adopted symmetry-breaking constraints (cf. Figure 1) needs to be
32 carefully chosen. As an example, DLV performs much better on the Ramsey
33 problem with LI symmetry-breaking constraints, but such performance is not
34 maintained when the more complex SB schema is adopted. A similar behavior
35 can be observed on SMOBELS and CMOBELS.

36 As for Social golfer, Table 3 does not show significant performance improve-
37 ments when symmetry-breaking is applied, with the ASP solvers (especially
38 SMOBELS) being significantly slowed down when adopting the most complex
39 schemas (LX and L2). However, it is worth noting that, on smaller negative
40 (non-benchmark) instances, impressive speed-ups have been obtained for all
41 systems, especially when using SA. As for LX, we also observe that it can be
42 applied in two different ways, i.e., forcing either players' schedulings or weekly
43 groupings to be lexicographically ordered.¹³ Values reported in Table 3 are
44 obtained by lexicographically ordering weekly groupings: as a matter of fact,
45 ordering players' schedulings is even less efficient on SMOBELS, being compar-
46 able for the other solvers. General rules for determining the right "amount" of
47 symmetry breaking for any given solver on different problems are currently still
48 unknown, but it seems that the simplest ones like SA or LI (cf. Figure 1) have
49 to be preferred when using ASP solvers. On the other hand, from the experi-
50 ments it results that OPL may benefit also from the addition of more complex
51 schemas, like LX (cf., e.g., Word design). This is likely due to the more robust
52 algorithms available in OPL for dealing with numbers and arithmetics (cf. also
53

54
55 ¹³None of the systems provide built-in constructs for expressing lexicographical orderings:
56 hence, they have been modelled as ordinary constraints.
57
58

1
2
3
4
5
6
7
8
9 the forthcoming paragraph on this subject).

10
11 **Impact of global constraints.** Experiments confirm that OPL may benefit
12 from the use of global constraints. As an example, the base specification of
13 the Golomb rulers problem encodes the constraint that forces the differences
14 between pairs of marks to be all different by a set of binary inequalities. By
15 replacing them with an `alldifferent` constraint, OPL was able to solve the
16 instance with 11 marks in the time-limit, and time required to solve smaller
17 instances significantly decreases. Also the Social golfer specification can be
18 restated by using global constraints, in particular the `distribute` constraint.
19 However, in this case our results show that OPL does not benefit from such
20 a reformulation, in that it was not able to solve even the 4-weeks instance
21 (solved in about 11 seconds with the base specification). Global constraints
22 help OPL also on other problems, e.g., All-Interval series (`alldifferent`) and
23 Car sequencing (`distribute`), even if, for the latter problem, the performance
24 improvements don't make it able to solve larger instances. Finally, Word design
25 seems not to be affected by the introduction of `distribute`.
26
27

28 **Impact of auxiliary predicates.** As already mentioned, auxiliary predicates
29 increase the size of the search space, but in some cases their use may result in
30 better performance, especially when their introduction leads to great simplifica-
31 tions of complex constraints. Actually, experiments show that, almost always,
32 ASP solvers benefit from the use of auxiliary predicates (often not really needed
33 by OPL, which allows to express more elaborate constraints), especially when
34 they are defined relying on the minimal model semantics of ASP (hence, in rule
35 heads). Good examples are Golomb rulers, where adding the auxiliary pred-
36 icate `difference/2` leads to a strong improvement of all solvers except DLV,
37 Water buckets, Langford numbers (where positive instances are solved much
38 faster by all systems considered), All-interval, and the Social golfer problem,
39 where SMOBELS is able to solve the 6-weeks instance in 6 seconds when the
40 auxiliary `meet/3` predicate is used, while solving the base specification requires
41 41 minutes (a similar behavior is observed with the other solvers). Using the
42 `meet` auxiliary predicate in Social golfer helps also OPL (but only after a sim-
43 ple search procedure that excludes branches on its variables has been added,
44 cf. Section 3.2). In particular, the 5-weeks instance has been solved in just 8
45 seconds (with respect to the 80 seconds of the base specification). It can be
46 observed that only in very few cases solvers suffer from the addition of auxiliary
47 predicates (cf., e.g., results of the Maximum density still life problem).
48
49

50 **Synergic reformulations.** Specifications obtained by combining, for each
51 problem and solver, the most two efficient techniques, in many cases further
52 boost performance, or at least does not affect it negatively. This is the case of,
53 e.g., Golomb rulers, Social golfer, Car sequencing, Word design (OPL and SMO-
54 BELS), Magic squares, Langford numbers, and All-interval series (CMODELS), that
55 often proved to be able to deal with larger instances, or to be able to run signif-
56
57
58

1
2
3
4
5
6
7
8
9 icantly faster. Few exceptions do exist, e.g., Maximum density still life, where
10 solving times were not significantly influenced. This gives evidence that combin-
11 ing “good” reformulations is in general a promising strategy to further boost
12 performance of all solvers.

13 Tables 2 and 3 also show some of the results obtained by other possible
14 combinations, without considering any uniformity criteria. It can be observed
15 that in few cases even better results could be achieved (cf., e.g., OPL on Golomb
16 rulers or All-interval series and the specification with auxiliary predicates, global
17 constraints and symmetry-breaking, or Social golfer and the specification with
18 auxiliary predicates and global constraints), but in several others, only worse
19 results were obtained.
20

21 **Impact of numbers and arithmetic constraints.** As introduced at the
22 end of Section 3.1, we evaluated the impact of numbers and arithmetics on the
23 performance of the various solvers on two problems, Water buckets and Magic
24 squares. In particular, we built new instances, equivalent to the original ones,
25 but such to force the various solvers to deal with larger numbers, and measured
26 the degradation of performance obtained by the most efficient specifications. In
27 detail, we proceeded as follows:
28

- 29 • As for Water buckets, we computed the new instances by doubling both
30 the buckets capacities and the water contents in the start and goal states;
31
- 32 • As for Magic squares, we changed the domain of values for each cell from
33 the interval $[1, n]$ (n being the board size) to the set $\{i \cdot k \mid i \in [1, n]\}$ with
34 k being 2 or 3.
35

36
37 We then compared the time needed by the different solvers to solve all in-
38 stances (for what concerns Water buckets) and the largest instance reported in
39 Table 3 (for what concerns Magic squares) with the most efficient specification
40 (that with auxiliary predicates), with the time needed to deal with the instance
41 obtained by performing the above modifications (to ensure fairness, no timeout
42 has been enforced in this case).

43 Results are shown in Table 4. From there, it can be observed that, as
44 expected, the presence of larger numbers and arithmetic constraints over them is
45 a major obstacle for all solvers. However, from the results it follows that globally,
46 OPL behaves much better than the ASP solvers on these issues, presumably
47 because of the more powerful arc consistency techniques provided by this solver.
48 Nonetheless, we must also pinpoint the good overall performance of CMODELS
49 with respect to the other ASP solvers, which is likely to depend on the higher
50 performance of current SAT algorithms already discussed above.
51

52 5 Conclusions

53
54 In this paper we reported results about an experimental investigation which
55 aims at comparing the relative efficiency of a commercial backtracking-based
56
57

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).

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 like symmetry-breaking, the adoption of global constraints and the addition of auxiliary predicates, 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 C MODELS 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.

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. However, even if our experiments suggest some good modelling practices, an exact understanding of which reformulations lead to the best performance for a given problem and solver remains a challenge.

Finally, we also performed an investigation about the impact of numbers

1
2
3
4
5
6
7
8
9 and arithmetic constraints in problem specifications, and highlighted the better
10 overall behavior of OPL with respect to ASP solvers in this context. However,
11 also about this issue, the efficiency of the SAT approach emerges, with the worse
12 performance of DLV and SMOBELS with respect to CMOBELS.
13

14 References

- 15
16
17 [1] M. Cadoli and T. Mancini. Exploiting functional dependencies in declarative
18 problem specifications. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 628–640, Lisbon, Portugal, 2004. Springer.
19
20
21 [2] M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. *Artificial Intelligence*, 170(8–9):779–801, 2006.
22
23 [3] M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. *Applied Artificial Intelligence*, 21(3), 2007. Special issue on “Best papers from AI*IA 2005”. To appear.
24
25 [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.
26
27 [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.
28
29 [6] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.
30
31 [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.
32
33 [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.
34
35 [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.
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9 [10] A. K. Dewdney. The game life acquires some successors in three dimensions. *Scientific American*, 224(2):112–118, 1987.
- 11
12 [11] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and
13 ASP solutions to NP-complete problems. In *Proceedings of the Twentyfirst*
14 *International Conference on Logic Programming (ICLP 2005)*, volume 3668
15 of *Lecture Notes in Computer Science*, pages 67–82, Sitges, Spain, 2005.
16 Springer.
- 17
18 [12] T. Ellman. Abstraction via approximate symmetry. In *Proceedings of the*
19 *Thirteenth International Joint Conference on Artificial Intelligence (IJ-*
20 *CAI'93)*, pages 916–921, Chambéry, France, 1993. Morgan Kaufmann, Los
21 Altos.
- 22
23 [13] A. J. Fernández and P. M. Hill. A comparative study of eight constraint
24 programming languages over the Boolean and Finite Domains. *Constraints*,
25 5(3):275–301, 2000.
- 26
27 [14] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and
28 T. Walsh. Breaking row and column symmetries in matrix models. In *Pro-*
29 *ceedings of the Eighth International Conference on Principles and Practice*
30 *of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in*
31 *Computer Science*, page 462 ff., Ithaca, NY, USA, 2002. Springer.
- 32
33 [15] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language*
34 *for Mathematical Programming*. International Thomson Publishing, 1993.
- 35
36 [16] E. C. Freuder and D. Sabin. Interchangeability supports abstraction and
37 reformulation for multi-dimensional constraint satisfaction. In *Proceedings*
38 *of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*,
39 pages 191–196, Providence, RI, USA, 1997. AAAI Press/The MIT Press.
- 40
41 [17] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelli-*
42 *gence*, 57:323–389, 1992.
- 43
44 [18] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scar-
45 cello. The DLV System for Knowledge Representation and Reasoning. *ACM*
46 *Transactions on Computational Logic*, 7(3):499–562, 2006.
- 47
48 [19] Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver
49 enhanced to non-tight programs. In V. Lifschitz and I. Niemelä, editors,
50 *Proceedings of the Seventh International Conference on Logic for Program-*
51 *ming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *Lec-*
52 *ture Notes in Computer Science*, pages 346–350, Fort Lauderdale, FL, USA,
53 2004. Springer.
- 54
55 [20] F. Lin and Z. Yuting. ASSAT: Computing answer sets of a logic program
56 by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.
- 57
58
59
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9 [21] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning
10 on problem specifications. In *Proceedings of the Sixth International Sym-*
11 *posium on Abstraction, Reformulation and Approximation (SARA 2005)*,
12 volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 165–181, Airth
13 Castle, Scotland, UK, 2005. Springer.
- 14
15 [22] I. Niemelä. Logic programs with stable model semantics as a constraint
16 programming paradigm. *Annals of Mathematics and Artificial Intelligence*,
17 25(3,4):241–273, 1999.
- 18
19 [23] N. Pelov, E. De Mot, and M. Denecker. Logic Programming approaches for
20 representing and solving Constraint Satisfaction Problems: A comparison.
21 In M. Parigot and A. Voronkov, editors, *Proceedings of the Seventh Inter-*
22 *national Conference on Logic for Programming and Automated Reasoning*
23 *(LPAR 2000)*, volume 1955 of *Lecture Notes in Computer Science*, pages
24 225–239, Reunion Island, FR, 2000. Springer.
- 25
26 [24] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction
27 problems. In H. J. Komorowski and Z. W. Ras, editors, *Proceedings of the*
28 *Seventh International Symposium on Methodologies for Intelligent Systems*
29 *(ISMIS'93)*, volume 689 of *Lecture Notes in Computer Science*, pages 350–
30 361, Trondheim, Norway, 1993. Springer.
- 31
32 [25] A. Ramani, F. A. Aloul, I. L. Markov, and K. A. Sakallak. Breaking
33 instance-independent symmetries in exact graph coloring. In *Proceedings*
34 *of Design Automation and Test Conference in Europe (DATE 2004)*, pages
35 324–331, Paris, France, 2004. IEEE Computer Society Press.
- 36
37 [26] J.-C. Régim. Global constraints and filtering algorithms. In M. Milano, edi-
38 tor, *Constraint and Integer Programming – Toward a Unified Methodology,*
39 *Operations Research/Computer Science Interfaces*, volume 27, chapter 4.
40 Kluwer Academic Publisher, 2003.
- 41
42 [27] O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.-H. Vu, and T.-V. Nguyen.
43 Benchmarking global optimization and constraint satisfaction codes. In
44 *Proceedings of the First International Workshop on Global Constraint Op-*
45 *timization and Constraint Satisfaction (COCOS 2002)*, volume 2861 of *Lec-*
46 *ture Notes in Computer Science*, pages 211–222, Valbonne-Sophia Antipo-
47 lis, France, 2003. Springer.
- 48
49 [28] B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and
50 implied constraints to model non-binary problems. In *Proceedings of the*
51 *Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*,
52 pages 182–187, Austin, TX, USA, 2000. AAAI Press/The MIT Press.
- 53
54 [29] G. Smolka. The Oz programming model. In *Computer Science Today: Re-*
55 *cent Trends and Developments*, volume 1000 of *Lecture Notes in Computer*
56 *Science*, pages 324–343. Springer, 1995.
- 57
58
59
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9 [30] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1988.
10
11 [31] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
12
13
14 [32] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On benchmarking constraint logic programming platforms. response to [13]. *Constraints*, 9(1):5–34, 2004.
15
16
17
18 [33] 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.
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65