Evaluating ASP and commercial solvers on the CSPLib
(Preliminary work)

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

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
cadoli|tmancini|micaletto|patrizi@dis.uniroma1.it

Abstract. This paper deals with three solvers for combinatorial problems: the commercial state-of-the-art solver Ilog OPL, and the research ASP systems DLV and SMODELS. The first goal of this research is to evaluate the relative performance of such systems, 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 selecting instances. The second goal is to analyze the effects of a popular reformulation technique, i.e., symmetry breaking, and the role of other modelling aspects, like global constraints and auxiliary predicates in constraint expressions.

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 and programs (complete, such as backtracking, or incomplete, such as tabu search, simulated annealing, etc.) or translating in a format suitable for Integer Programming solvers (e.g., Ilog CPLEX[1]), has been challenged by the use of libraries for Constraint Programming (CP), e.g., Ilog SOLVER[2], CLP(FD) [7], interfaced through general purpose programming languages such as C++, Java, or Prolog. At the same time, the need for a higher level of abstraction led to the design and development of 1) General purpose constraint modelling and programming languages (e.g., OPL [23], XPRESS^{MP}[3], GAMS [5], etc.), and 2) Languages based on specific solvers, such as AMPL [12] (an interface to mathematical programming solvers), DLV [13], SMODELS [17], ASSAT [15] (which translate into an instance of ASP – Answer Set Programming), NP-SPEC [4] (which translates into an instance of SAT).

This paper focuses on the last class of solvers, which are highly declarative, and characterized by the possibility of decoupling the specification of a problem from the instance, and by having optional procedural information. In particular, we consider one commercial state-of-the-art solver, i.e., Ilog OPL and some ASP solvers, namely DLV and SMODELS. The latter has the interesting property of sharing the specification language with several other solvers through the common parser LPARSE[4]. As a matter of fact, such systems exhibit interesting differences, including availability (OPL and ASP are, respectively, payware and freeware systems, the latter often being open source), algorithm used by the solver (resp. backtracking- and fixpoint-based), 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), possibility of specifying a separate search procedure.

The first goal of this research is to evaluate the relative performance of such systems, using a reproducible and extensible experimental methodology. In particular, we consider a third-party problem library, i.e., the CSPLib[5], and uniform rules for modelling and instance selection. *The second goal is to analyze the effects of* a popular reformulation technique, i.e., *symmetry breaking*, and the *role of* other modelling aspects, like *global constraints and auxiliary predicates*.

[1] cf. http://www.ilog.com/products/cplex.
[2] cf. http://www.ilog.com/products/solver.
[3] cf. http://www.dashoptimization.com.
[4] cf. http://www.tcs.hut.fi/Software/smodels.
[5] cf. http://www.csplib.org.

As for 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., [8, 11]). Since previous studies [19] 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 performances. 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 cite [10] and [24] where SOLVER is compared to other CP languages such as, e.g., OZ [22], CLAIRE[6], and various Prolog-based systems. Moreover, some benchmark suites have been proposed, cf., e.g., the COCONUT one [20]. Also on the ASP side, which has been the subject of much research in the recent years, some benchmark suites have been proposed in order to facilitate the task of evaluating improvements of their latest implementations, the most well-known being ASPARAGUS[7], and ASPLib[8]. 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 (cf., e.g., [9, 18]). In Section 5 we briefly relate our work with such papers.

In this research we consider the CSPLib problem library for our experiments. CSPLib is a collection of 45 problems, and is widely known 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. Since many of them are described only in natural language, this work also provides, as a side-effect, formal specifications of such problems in the modelling languages adopted by some solvers. In related work [3], we model a number of problems from the same library in the language NP-SPEC [4]. Ultimately, we plan to provide (and eventually make publicly available) specifications for several solvers, covering a large part of the CSPLib collection.

2 Methodology

In this section we present the methodology adopted in order to achieve the two goals mentiond in Section 1. For each problem considered we define a number of different formulations. The first one, called the *basic specification*, is obtained by a straightforward and intuitive "translation" of the CSPLib problem description into the target language. Arguably, the basic specification is the most natural and declarative that the user can write.

However, it is well-known that such a model is often not performant, and additional care is usually needed in order to improve its efficiency. To this end, the literature proposes several techniques: in this paper we mainly focus on performing symmetry-breaking, but make also some discussions about the role of global constraints and auxiliary predicates. Other reformulation techniques, e.g., the addition of implied constraints (cf., e.g., [21]), safe-delay of constraints [1], and the addition of procedural aspects, e.g., search strategies, are subject of current work.

As for symmetry-breaking, for each problem we identify its structural symmetries, i.e., those that depend on the problem structure, and not on the particular instance (along the lines of [16]), and exploit them by adding further constraints to the specification (the so-called symmetry-breaking constraints). Since symmetries can be broken in several ways, we define different high-level symmetry-breaking schemas, again in the spirit of [16], and write the corresponding models (cf. forthcoming Section 3.2).

We tried to do the modelling task in a way as systematic as possible, by requiring the specifications for the various solvers to be similar to each other: in particular, all the basic models share the same idea for the definition of the search space and the constraints, and also symmetry-breaking is addressed by following the same schemas. However, it is well-known that efficiency improvements can be obtained by lowering the degree of declarativeness of problem specifications,

[6] cf. `http://claire3.free.fr`.
[7] cf. `http://asparagus.cs.uni-potsdam.de`.
[8] cf. `http://dit.unitn.it/~wasp`.

and by exploiting the particularities of the different solvers. To this end, the "best" specification than can be written for each solver (obtained by ignoring the above mentioned declarativeness and uniformity criteria) should also be considered. However, such topic will not be discussed in this paper, being the subject of ongoing work.

As for the instances, in this paper we opted for problems whose input data is made of few integer parameters (with the exception of the Car sequencing problem, which will be discussed in Section 3.1). In order to have a synthetic qualitative measure of the performance of the various solvers, for each problem we fix all the input parameters but one. Hence in our results we report for each problem and each solver the largest instance (denoted by the value given to the selected parameter) that is solvable in a given time limit (one hour).

3 The experimental framework

3.1 Selecting the problems

So far, we have formulated and solved 4 problems, described in what follows, along with their identification number in CSPLib, the areas they belong to, and the single parameter that encodes the problem instances.

Ramsey problem (#017: Bin pack. and partitioning, Comb. math.).
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 values given to n.

Social golfer (#010: Sched., Bin pack. and part., 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 w weeks, such that no two golfers play in the same group more than once. Problem instances are denoted by values given to w, the schedule length.

Golomb rulers (#006: Frequency assign., Comb. math.).
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. Hence, instances of the problem are denoted by values given to m.

Car sequencing (#001: Scheduling).
A number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). 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 encoded by a single parameter. To this end, we considered some benchmarks suggested in the CSPLib, namely "4/72", "6/76" and "10/93". Actually, such instances are too hard for our solvers, hence we proceeded as follows: from any original benchmark (with n classes), we generated a set of instances by reducing the number of classes to all values in $[1, n]$, and consequently resizing station capacities in order to avoid an undesirable overconstraining that makes instances unfeasible. Thus, instances derived from the same benchmark can be ordered according to the value for the (reduced) number of classes, which can be regarded as a measure for their size.

3.2 Selecting the problem models

As already mentioned in Section 2, problem specifications have been derived following general and uniform criteria. Some technical details are given in the following.

General criteria. The first obvious difference between OPL and the ASP solvers concerns the search space declaration. The former language relies on the notion of *function* from a finite domain

to a finite domain, 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 DLV.

Secondly, since the domain itself can play a major role in efficiency, sometimes it has been inferred through some *a posteriori* consideration. As an example, in the Golomb rulers problem the maximum position for marks is upperbounded by 2^m, but choosing such a large number can advantage OPL, which has powerful arc-consistency algorithms for domain reduction. As a consequence, we used the domain $3L/2$ for all solvers, where L is the maximum mark value for the optimum of the specific instance.

Finally, since the performance typically depends on the instances being positive or negative, we considered the *optimization* versions of Ramsey and Golomb ruler problems. As a matter of facts, for proving that a solution is optimal, solvers have to solve both positive and negative instances. As for Social golfer, since the optimum (which, according to the literature, is either 9 or 10 weeks) cannot be achieved by our solvers, we dealt with its decisional version.

Understanding the role of symmetry-breaking. Symmetries are dealt with in a systematic way, by relying on general schemas of symmetry-breaking constraints already discussed in [16], which are briefly recalled in the following. In principle, a set of symmetries for a problem can be broken in several ways. In this work, we focus on the following general schemas (examples below are given in the simple case in which all permutations of values are symmetries, but generalizations do exist):

- **Selective assignment (SA):** A subset of the variables are assigned to precise domain values: a good example is in the Social golfer problem, where, 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: an example is given by the Golomb rulers problem, where, 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 are required to be such that, for any pair of values d and d', with $d < d'$, the least variable set to d is less than (wrt the order given on variables) the least variable set to 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 edge colored by red has an index lower than the least edge colored by blue, which in turn has an index lower than the least one colored by green.
- **Size-based ordering (SB):** In this case, after fixing a linear order on values, we force assignments to be such that the number of variables which are assigned value d are less than or equal to the number of variables which are assigned value d', for every $d \leq d'$. As an example, in the Ramsey problem, we force to color in green at least the number of edges colored in blue, which in turn must be greater than or equal to the number of edges colored in red. Generalizations of this schema do exist, in which the partitions of the set of variables into sets whose sizes are forced to be ordered are defined in different ways.
- **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):** 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., [11]). A good example is Social golfer, in which the search space can be defined as a 2D matrix that assigns a group to every combination player/week. 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).

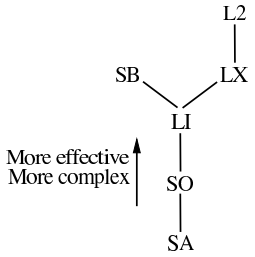

Fig. 1. Classification among symmetry-breaking schemas as a partial ordered set.

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 much" complex is their evaluation. Figure 1(d) shows such a classification as a partial order set: $s_1 < s_2$ iff s_2 reduces better the search space. However, s_2 typically requires more complex constraints.

Understanding the role of 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 `xi` $\neq$ `xj` (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. As a starting point for the evaluation of their impact, some reformulations of the base specifications are provided in terms of the OPL constructs `alldifferent` and `distribute`. According to the problems structure, the former has been applied to Golomb Rulers while the latter to Social Golfer and Car Sequencing. As for Ramsey Problem none of such reformulations can be given.

Since ASP solvers do not offer such a feature, no comparison can be made on this issue.

Understanding the role of auxiliary predicates. A guessed predicate 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., [6, 2]). Although, in general, the use of auxiliary predicates increases the size of the search space, in some cases this results in a simplification of complex constraints and in a reduction of the number of their variables, and hence may lead to appreciable time savings.

We consider equivalent specifications for the Ramsey and Social golfer problems, obtained by using auxiliary predicates. In particular, we define the auxiliary `color_used/1` predicate (with an obvious semantics) in the specification for the former problem, and the predicate `meet/3` for the latter one, with tuples $\langle p_1, p_2, w\rangle$ meaning that players p_1 and p_2 join the same group on week w.

Figure 2(a)–(c) shows the base specifications for the Ramsey problem written in all languages, and Figure 2(d)–(f) presents the LI symmetry-breaking constraints that can be added in order to break the permutation symmetry on colors. It is worth noting that, in both Ramsey Problem and Social Golfer, the use of the auxiliary predicate `color_used()` seems unavoidable for SMODELS and, therefore, their Base specification and that with the auxiliary predicate coincide for such solver.

Furthermore, some specifications used for the Golomb rulers problem are presented in Figure 3(a)–(c), as well as SO symmetry-breaking constraints that force the difference between the first two marks to be less than that between the last two. In particular, the search space for the Golomb ruler problem specifications is defined to be a total function *ruler* from marks $[1, m]$ to positions, i.e., $ruler : [1, m] \rightarrow [0, maxLength]$, where $maxLength$ is set a-posteriori to be $3L/2$, as discussed at the beginning of this subsection. Of course, additional constraints are needed in the ASP specifications in order to force the *ruler* predicate to be mono-valued. As for the constraints,

OPL *base specification*

```
int+ nnodes = ...;   int+ ncolors = ...;
range colors 1..ncolors;  range nodes 1..nnodes;
struct edge { nodes n;   nodes m;  };
{edge} edges = {<n,m> | n,m in nodes : n < m};

var colors coloring[edges];
minimize (sum (c in colors)
  ((sum (e in edges) (coloring[e] = c)) > 0))
subject to {
  forall(x,y,z in nodes: (x < y < z)) { // c1
     not (coloring[<x,y>] = coloring[<x,z>] =
          coloring[<y,z>]);
  };
};
```

(a)

DLV *base specification*

```
coloring(N,M,C) v fail(N,M,C) :- colors(C), nodes(N),
                                 nodes(M), N<M.
:- not #count{C: coloring(N,M,C) }=1, nodes(N),
   nodes(M), N<M.

:- coloring(X,Y,C), coloring(Y,Z,C), coloring(X,Z,C). % c1

:~  #count{A,B: coloring(A,B,C)}>0, allColors(C). [1:1]
```

(b)

SMODELS *base specification*

```
1{ coloring(N,M,C) : colors(C) }1 :- nodes(N;M), lt(N,M).
:- nodes(X;Y;Z), gt(Y,X), gt(Z,Y), coloring(X,Y,C), % c1
   coloring(Y,Z,C), coloring(X,Z,C), colors(C).
color_used(C) :- coloring(N,M,C), colors(C),
                 nodes(N;M), lt(N,M).
minimize{ color_used(C): colors(C) }.
```

(c)

OPL *LI symmetry-breaking constraint*

```
forall(<f1,t1>, <f2,t2> in edges :
  f1<f2 \/ (f1=f2 & t1<t2)) {
    coloring[<f1,t1>] > coloring[<f2,t2>] => (
      sum(<f3, t3> in edges) (
        (f3<f1 \/ (f3=f1 & t3<t1)) &
        (coloring[<f3, t3>] = coloring[<f2,t2>]) ) > 0
    );
}
```

(d)

DLV *LI symmetry-breaking constraint*

```
minEdge(C,X,Y) :-
  #min{N: coloring(N,M,C) }=X,
  #min{M: coloring(X,M,C) }=Y,
  colors(C).
:- minEdge(C1,X1,Y1), minEdge(C2,X2,Y2),
   C1<C2, X2<X1.
:- minEdge(C1,X1,Y1), minEdge(C2,X2,Y2),
   C1<C2, X2==X1, Y2<Y1.
```

(e)

SMODELS *LI symmetry-breaking constraint*

```
1{ minEdge(C,X,Y): nodes(X;Y): lt(X,Y)}1 :- colors(C).
:- minEdge(C,X,Y), not coloring(X,Y,C),
   nodes(X;Y), lt(X,Y), colors(C).
:- minEdge(C,X,Y), coloring(X1,Y1,C),
   or(lt(X1,X), and(eq(X,X1),lt(Y1,Y))),
   nodes(X;X1;Y;Y1), colors(C).
:- minEdge(C,X,Y), minEdge(C1,X1,Y1), gt(C1,C),
   or(lt(X1,X), and(eq(X,X1),lt(Y1,Y))),
   nodes(X;X1;Y;Y1), colors(C;C1).
```

(f)

Fig. 2. (a), (b), (c): Base specifications for the Ramsey problem in all languages. (d), (e), (f): LI symmetry-breaking constraints added to the respective specifications.

c1 forces the first mark to be put at position 0, while *c2* forces marks to be put from left to right in ascending order. *c3* constrains the differences between marks to be all different. It is worth noting that, in order to express the last constraint, we can use the global constraint `alldifferent` in OPL (cf. Figure 3(a)), and define and use the auxiliary predicate `diff` in the ASP specifications (cf. Figure 3(b) and (c)).

4 Experimental results

Our experiments have been performed by using the following solvers: *i)* Ilog SOLVER v. 5.3, invoked through OPLSTUDIO 3.61, *ii)* SMODELS v. 2.28, by using LPARSE 1.0.17 for grounding, *iii)* DLV v. 2005-02-23, on a 2 CPU Intel Xeon 2.4 Ghz computer, with a 2.5 GB RAM and Linux v. 2.4.18-64GB-SMP.

For every problem, we wrote the specifications described in Sections 2 and 3 in the different languages. We then ran the different specifications for each solver on the same set of instances, with a timeout of one hour. Table 1 shows a summary of the results concerning base specifications and their various reformulations by symmetry-breaking. In particular, for each problem and solver, we report the largest instance that the various systems were able to solve (in the given time-limit) for the different specifications. As for Car sequencing, we report, for each set of instances generated from CSPLib benchmarks "4/72", "6/76" and "10/93", the largest one solved, i.e., the one with the largest number of classes (cf. the discussion about instance selection for this problems in Section 3.1).

Problems	Solvers														
Ramsey Problem	OPL						DLV					SMODELS			
	Base	LI	SB	Aux			Base	LI	SB	Aux		Base/Aux*	LI	SB	
(max no. of nodes)	**16**	13	**16**	**16**			9	14	8	9		9	**16**	8	
Social Golfer	OPL						DLV					SMODELS			
	Base	SA	L2	LX	Aux	GC	Base	SA	L2	LX	Aux	Base/Aux*	SA	L2	LX
(max no. of weeks)	5	5	5	5	5	3	4	**6**	**6**	**6**	**6**	**6**	**6**	0	5
Golomb Rulers	OPL						DLV					SMODELS			
	Base	SO	Aux	GC			Base	SO	Aux			Base	SO	Aux	
(max no. of marks)	10	10	**11**	**11**			9	9	9			6	6	8	
Car Sequencing	OPL						DLV					SMODELS			
(max no. of classes)	Base	SO	Aux	GC			Base	SO	Aux			Base	SO	Aux	
bench. 4/72	10	10	10	10			**13**	**13**	**13**			**13**	**13**	**13**	
	Base	SO	Aux	GC			Base	SO	Aux			Base	SO	Aux	
bench. 6/76	6	6	6	6			**9**	**9**	**9**			**9**	**9**	**9**	
	Base	SO	Aux	GC			Base	SO	Aux			Base	SO	Aux	
bench. 10/93	**12**	**12**	**12**	**12**			**12**	**12**	**12**			**12**	**12**	**12**	

* As for Ramsey and Social Golfer problems, the use of the auxiliary predicate seems unavoidable in the SMODELS specification (cf. Figure 2(c)). To this end, the Base and Aux specification coincide.

Table 1. Sizes of the largest instances solved by OPL, DLV, and SMODELS in 1 hour, using the base specification and their reformulations by symmetry-breaking. Bold numbers denote the best results.

Moreover, in Figure 4 we also show also the evolution of solution time wrt the size of the instance for two problems, i.e., Ramsey and Golomb rulers, when using the various specifications on the different solvers.

Results are still preliminary, but make it possible to get some interesting analyses.

Role of symmetry-breaking. From the experiments, it can be observed that symmetry-breaking may prove to be beneficial, 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 it is slowed down when the more complex SB schema is adopted. A similar behavior can be observed by SMODELS.

As for Social golfer, Table 1 does not show significant performance improvements when symmetry-breaking is applied to OPL or SMODELS. However, when solving smaller negative (non-benchmark) instances, impressive speed-ups have been obtained for all systems, especially when using the simpler SA schema. As for the LX symmetry-breaking constraints for this problem, we observe that it can be applied in two different ways, i.e., forcing either players' schedulings or weekly groupings to be lexicographically ordered. Values reported in Table 1 are obtained by lexicographically ordering weekly groupings: as a matter of facts, ordering players' schedulings is less performant on SMODELS, 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.

Role of global constraints. Experiments confirm that OPL may benefit from the use of global constraints. As an example, by replacing in the Golomb rulers problem the `alldifferent` constraint among differences between pairs of marks by a set of binary inequalities, OPL is not able to solve the instance with 11 marks in the time-limit, and time required to solve smaller instances increases. Also the Social golfer specification can be restated by using global constraints, in particular the `distribute` constraint. However, in this case our preliminary results show that OPL does not benefit from such a reformulation, in that it was not able to solve the 5-weeks instance (solved in about 80 seconds with the base specification).

Currently, we are performing further experiments in order to understand the effectiveness of using global constraint in other problems. As an example, also Car sequencing can be reformulated by using `distribute`.

OPL *specification with global constraint*

```
int+ m = ...;
int+ maxLength = ...;
range marks 1..m;
range positions 0..maxLength;
var int+ ruler[marks] in positions;

minimize ruler[m]
subject to {
   ruler[1] = 0;  // c1
   forall (i in 1..m-1) {  // c2
      ruler[i] < ruler[i+1];
   };
   alldifferent(
      all(i,j in 1..m: i <> j)
            (ruler[j] - ruler[i])); // c3
};
```

(a)

DLV *specification with aux. predicate*

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

ruler(1,0).     % c1
:- ruler(M1,P1), ruler(M2,P2), M2>M1, P2<=P1. % c2

% c3
diff(M1,M2,D) :- ruler(M1,P1), ruler(M2,P2),
                 M2>M1, P2=D+P1.
:-  #count{M1,M2 : diff(M1,M2,D)}>1, marks(D).

:~ ruler(M,P), m(P). [P:1]
```

(b)

SMODELS *specification with aux. predicate*

```
1 {ruler(M,P): positions(P)} 1 :- marks(M).

minimize[ruler(m,P): positions(P)].

compute {ruler(1,0)}.                  % c1
:- ruler(M,P), ruler(M1,P1), gt(M1,M),
   not gt(P1,P).                       % c2

diff(M1,M2,D) :- ruler(M1,P1), ruler(M2,P2), % c3
                 gt(M2,M1), D=P2-P1.
:- 2{diff(P1,P2,D): positions(P1;P2): gt(P2,P1)}, marks(D).
```

(c)

OPL *SO symmetry-breaking constraint*

```
((ruler[2]-ruler[1]) < (ruler[m]-ruler[m-1]));
```

(d)

DLV *SO symmetry-breaking constraint*

```
:- diff(1,2,D1), m(M), M=M1+1,
   diff(M1,M,D2), D2<D1.
```

(e)

SMODELS *SO symmetry-breaking constraint*

```
:- diff(1,2,D1), diff(m-1,m,D2),
   lt(D2,D1), marks(D1;D2).
```

(f)

Fig. 3. (a), (b), (c): Some specifications for the Golomb rulers problem in all languages (with the `alldifferent` global constraint for OPL, and with the auxiliary predicate `diff` for ASP solvers). (d), (e), (f): SO symmetry-breaking constraints added to the respective specifications.

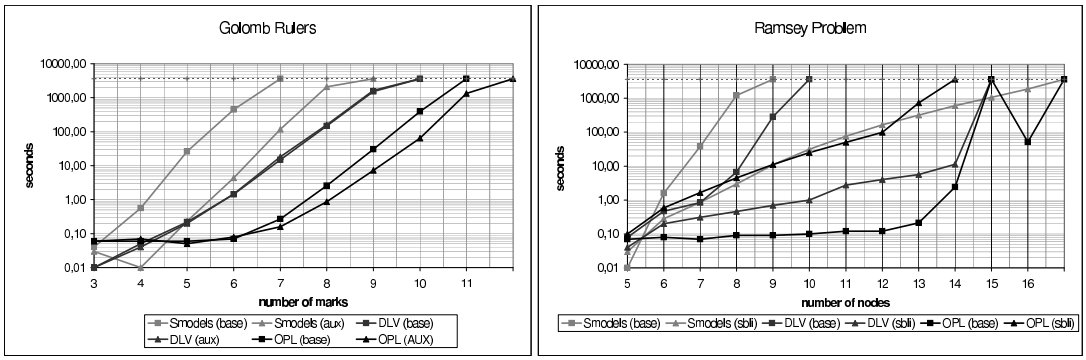

Fig. 4. Evolution of solution time in the size of the instance for Ramsey and Golomb rulers problems.

Role of auxiliary guessed predicates. ASP solvers seem to 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 the minimal model semantics of ASP (hence, in rule heads). As an example, SMODELS solves the 6-weeks instance of the Social golfer problem in 6 seconds, when the auxiliary `meet/3` predicate is used, while solving the base specification requires 41 minutes. Even if not as much remarkable, a similar behavior is observed in DLV. Similar results have been obtained for the Ramsey problem, where the auxiliary predicate `color_used/1` is used.

Using the `meet` auxiliary predicate helps also OPL. In particular, the 5-weeks instance has been solved in just 8 seconds (with respect to the 80 seconds of the base specification). It is interesting to note that, by further adding a suitable search procedure, solving time dropped down to less than a second. This is a good evidence that exploiting the peculiarities of the solver at hand may significantly increase the performance. As already stated in Section 1, this topic is subject of current work.

5 Conclusions, related and current work

In this paper we reported some preliminary results about an experimental investigation which aims at comparing the relative efficiency of a commercial backtracking-based and two academic ASP solvers. In particular, we modelled a number of problems from the CSPLib into the languages used by the different solvers, in a way as systematic as possible, by also applying symmetry-breaking and using global constraints and auxiliary guessed predicates.

As already observed in Section 1, not much work has been done in comparing solvers based on different technologies. To this end, we cite [9], 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 [18], where ASP and Abductive Logic Programming systems, as well as a first-order finite model finder, are compared in terms of modelling languages and relative performances on three problems: Graph coloring, N-queens, and a scheduling problem. The main difference of our work with respect to the above ones is the attempt to define and use a uniform metodology to select problem specifications and instances, and to emphasize the impact of different modelling issues. In particular, for each problem, a base specification and several reformulated ones have been proposed, with the aim of investigating the impact of different techniques (symmetry breaking, use of global constraints and of auxiliary predicates) on solvers' performances. The different approach we adopted led to different results: as an example, in [9] the ASP system SMODELS turns out to be always outperformed by the others. In our experiments, which rely on different problems, SMODELS seems to be competitive, even if not winning, on some of them, and to be positively affected by the proposed reformulation techniques (cf. the evolution of solution time for Ramsey and Golomb rulers problems depicted in Figure 4). Also in [18] the SMODELS behavior turns out to be unclear. It is worth noting that the system used in that paper does not allow the specification of optimization problems, which, in contrast, have been considered in the present work.

Our current efforts are mainly aimed at increasing the number of analyzed problems, in order to cover a large part of the CSPLib, following the methodology described in this paper, as well as to extend the analysis to other ASP systems, in particular CMODELS [14], which uses SAT, and is known to be very efficient in many cases (cf., e.g., [9]). Next, we plan to extend the set of refomulations by also exploiting: *(i) Safe delay* of constraints, *(ii) Implied constraints*, and *(iii)* The particularities of the various solvers, in order to take into account, for each problem, the best model that can be written, ignoring the uniformity and declarativeness issues discussed in Section 2. Such work will provide a large collection of problem formulations in several modelling languages, that we plan to make publicly available to the community, and will result in a good basis for a better understanding of good practices in modelling problems in different languages.

Finally, a future, and even more exhaustive extension of the collection can be obtained by including other systems, e.g., more ASP and backtracking-based solvers, or systems based on different technologies, like SAT-compilers and local search solvers.

References

1. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In *Proc. of KR 2004*, pages 388–398, Whistler, BC, Canada, 2004. AAAI Press/The MIT Press.
2. M. Cadoli and T. Mancini. Exploiting functional dependencies in declarative problem specifications. In *Proc. of JELIA 2004*, volume 3229 of *LNAI*, Lisbon, Portugal, 2004. Springer.
3. M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. In *Proc. of CILC 2005*, Roma, Italy, 2005.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162:89–120, 2005.
5. 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.
6. 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.
7. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *J. of Logic Programming*, 27:185–226, 1996.
8. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of KR'96*, pages 148–159, Cambridge, MA, USA, 1996. Morgan Kaufmann, Los Altos.
9. A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proc. of ICLP 2005*, volume 3668 of *LNCS*, pages 67–82, Sitges, Spain, 2005. Springer.
10. 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.
11. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proc. of CP 2002*, volume 2470 of *LNCS*, page 462 ff., Ithaca, NY, USA, 2002. Springer.
12. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. Intl. Thomson Publ., 1993.
13. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. on Comp. Logic*. To appear.
14. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver enhanced to non-tight programs. In V. Lifschitz and I. Niemelä, editors, *Proc. of LPNMR 2004*, volume 2923 of *LNCS*, pages 346–350, Fort Lauderdale, FL, USA, 2004. Springer.
15. F. Lin and Z. Yuting. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1–2):115–137, 2004.
16. T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proc. of SARA 2005*, volume 3607 of *LNAI*, pages 165–181, Airth Castle, Scotland, UK, 2005. Springer.
17. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artif. Intell.*, 25(3,4):241–273, 1999.
18. 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, *Proc. of LPAR 2000*, volume 1955 of *LNCS*, pages 225–239, Reunion Island, FR, 2000. Springer.
19. A. Ramani, F. A. Aloul, I. L. Markov, and K. A. Sakallak. Breaking instance-independent symmetries in exact graph coloring. In *Proc. of DATE 2004*, pages 324–331, Paris, France, 2004. IEEE Comp. Society Press.
20. O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.-H. Vu, and T.-V. Nguyen. Benchmarking global optimization and constraint satisfaction codes. In *Proc. of COCOS 2002)*, volume 2861 of *LNCS*, pages 211–222, Valbonne-Sophia Antipolis, France, 2003. Springer.
21. B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proc. of AAAI 2000*, pages 182–187, Austin, TX, USA, 2000. AAAI Press/The MIT Press.
22. G. Smolka. The Oz programming model. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.
23. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
24. M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On benchmarking constraint logic programming platforms. response to [10]. *Constraints*, 9(1):5–34, 2004.