SAT as an Effective Solving Technology for Constraint Problems

Marco Cadoli, Toni Mancini, and Fabio Patrizi

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
{cadoli, tmancini, patrizi}@dis.uniroma1.it

Abstract. In this paper we investigate the use of SAT technology for solving constraint problems. In particular, we solve many instances of several common benchmark problems for CP with different SAT solvers, by exploiting the declarative modelling language NPSpec, and Spec2Sat, an application that allows us to compile NPSpec specifications into SAT instances. Furthermore, we start investigating whether some reformulation techniques already used in CP are effective when using SAT as solving engine. We present encouraging experimental results in this direction, showing that this approach can be appealing.

1 Introduction

Several benchmark problems have been proposed in the literature for testing the performance of Constraint Programming tools e.g., OR-Library (`www.ms.ic.ac.uk/info.html`) or CSPLib (`www.csplib.org`), spanning several areas, from combinatorics, to planning, scheduling, or problems on graphs. There is also a great variety in the kind of solvers that are used for Constraint Programming: those based on backtracking (cf., e.g., [15]), mathematical programming (cf., e.g., [5]), answer sets and stable model semantics (cf., e.g., [8]), which are typically complete, or those that rely on local-search techniques (cf., e.g., [10]) which are intrinsically incomplete.

In this paper we focus on a different kind of tools, i.e., solvers for Propositional Satisfiability (SAT), showing how they can be transparently and effectively used for solving constraint problems. The intuition behind the usage of a SAT solver, is that every CSP can be reduced in polynomial time to an instance of SAT, since the complexity of solving a CSP is in NP, and SAT is one of the prototypical NP-complete problems. Actually, the latter aspect led to a great interest and to a huge amount of research in the field of SAT solving (cf., e.g., the proceedings of the last SAT conferences), leading to the current availability of very efficient solvers that can deal with very large formulae. State-of-the-art SAT solvers include complete ones, such as zChaff [11], and incomplete ones, such as walksat [14], and bg-walksat [16]. For an up-to-date list, we refer the reader to the URLs `www.satlib.org` and `www.satlive.org`.

The availability of fast solvers for SAT drew a great interest in the CP research community, and many papers show how to translate (compile) into SAT instances of various problems (cf., e.g., [7,6]). However, the complexity of the translation task is a major obstacle, since the compilation strongly depends on the constraints of the problem to be solved. Nowadays, this task is typically made by problem-dependent programs hence, in practice, preventing SAT to be one of the actual solving technologies for Constraint Programming. The availability of specification languages and systems that compile problem instances into SAT formulae, e.g., the language NPSpec and the Spec2Sat system [3], is an important step ahead, providing the user with the possibility of easily building specifications for new constraint problems in a purely declarative way, maintaining a strong independence on the instances.

In this paper, we present some experiments showing that SAT technology can be *effectively* employed for solving CSPs, by using NPSpec on several common benchmark problems for CP, experiencing different SAT solvers. The problems we focus on are a significant subset of those present in the benchmark repository CSPLib, very well-known in the CP research community. Problems in CSPLib are usually described only in natural language, and no formal specification is given for most of them. Hence, as a side-effect, our work also proposes declarative specifications (in the language NPSpec) for such problems.

In general, given a specification of a problem, several techniques have been proposed to reformulate it, in order to improve the solver efficiency, while maintaining equivalence (or at least, the possibility to efficiently reconstruct valid solutions to the original problem from solutions to the reformulated one). We experienced the application, at the symbolic level of the specification, of two of them, and present some experimental results.

The paper is organized as follows: in Section 2 we briefly illustrate the language NPSpec and the Spec2Sat program that, given a NPSpec specification and an instance, compiles it into a SAT instance. In Section 3 we present the chosen benchmark problems, while in Section 4 we present and comment our experimental results. Finally, Section 5 draws further discussions and concludes the paper.

2 The NPSpec Language and Spec2Sat

NPSpec and Spec2Sat have been extensively described in [3]. Hence, in what follows, we just recall the syntax and the informal semantics of the modelling language, and the general architecture of the compiler. The Home Page of the NPSpec project (`www.dis.uniroma1.it/~cadoli/research/projects/NP-SPEC/`) contains all the specifications proposed in this paper, as well as the program itself.

The NPSpec *language.* An NPSpec program consists of a `DATABASE` section and a `SPECIFICATION` section. The former includes the definition of the problem instance, in terms of extensional relations, and integer intervals and constants. The latter section instead, consists of the problem specification, that is divided into two parts: the declaration of a *search space*, and the definition of constraints that a point in the search space has to satisfy in order to be a solution to the problem instance. The declaration of the constraints is given by a stratified DATALOG program, which can include the six predefined relational operators and negative literals.

The full syntax of NPSPEC is given in [3], hence here we just recall it with an example. In particular, we show an NPSPEC program for the *Social golfer* NP-complete problem (problem nr. 10 in CSPLib), that, given a set of `N_PLAYERS` players that want to play golf once a week, amounts to find an arrangement for all of them into a number `N_GROUPS` of groups of size `GROUP_SIZE` for `N_WEEKS` weeks, in such a way that no two players play more than once in the same group. The following relationship must hold among three of the afore-mentioned quantities: `N_PLAYERS` = `N_GROUPS` ∗ `GROUP_SIZE`.

```
DATABASE
  N_WEEKS = 3;   N_GROUPS = 3;   GROUP_SIZE = 3;
  N_PLAYERS = N_GROUPS * GROUP_SIZE;
SPECIFICATION
  IntFunc({1..N_PLAYERS}><{1..N_WEEKS}, play, 1..N_GROUPS). // S1
  fail <-- play(P1,W1,G1), play(P2,W1,G1), P1 != P2,        // S2
           play(P1,W2,G2), play(P2,W2,G2), W1 != W2.
  fail <-- COUNT(play(*,W,G),X), X != GROUP_SIZE.           // S3
```

The following comments are in order:

– The input instance is defined in the `DATABASE` section, which is generally provided in a separate file (this part may also define finite relations).
– In the search space declaration (metarule `S1`) the user declares the predicate symbol `play` to be a "guessed" one, implicitly of arity 3. All other predicate symbols are, by default, not guessed. Being guessed means that we admit all extensions for the predicate, subject to the other constraints.
– `play` is declared to be an integer function from the finite domain {`1..N_PLAYERS`} × {`1..N_WEEKS`} to {`1..N_GROUPS`}. As an example, $\{(1,1,1),(2,1,1),(3,1,1),\quad(4,1,2),(5,1,2),(6,1,2),(7,1,3),(8,1,3),(9,1,3),\ldots\}$ is a valid extension as long as it defines exactly one group for any player in any week.
– Comments can be inserted using the symbol "`//`".
– Rules `S2` and `S3` are the constraints that schedulings must obey in order to be valid solutions: a scheduling `fail`s, i.e., it is not valid, if there exist two players `P1` and `P2` that play twice in the same group (rule `S2`), or if there exist a group `G` which is not of size `GROUP_SIZE` in a week `W` (rule `S3`).

Solving this program with NPSPEC produces an extension for the guessed predicates that satisfies all the constraints, if it exists.

The search space declaration, which corresponds to the definition of the domain of the guessed predicates, is, in general, a sequence of declarations of the form: *(i)* `Subset(<domain>, <pred_id>)`; *(ii)* `Permutation(<domain>, <pred_id>)`; *(iii)* `Partition(<domain>, <pred_id>, n)`;

(iv) `IntFunc(<domain>, <pred_id>, min..max).` We do not formally give further details of the NPSpec syntax, but, in the following sections, we present and comment several other examples. We just remark that the declarative style of programming in NPSpec is very similar to that of DATALOG, and it is therefore easy to extend programs for incorporating further constraints. Concerning syntax, we remark that NPSpec offers also useful SQL-style aggregates, such as `SUM`, `COUNT`, `MIN`, and `MAX`. Several examples of Section 3 use such operators.

The NPSpec *to SAT compiler.* Spec2Sat is an application that allows the compilation of a NPSpec specification (when given together with input data) into a SAT instance. We do not give here the technical details of the compilation task, that can be found in [3], but just briefly describe the general architecture of the application. Given two text files, containing the specification S in NPSpec and the instance data I, Spec2Sat compiles $S \cup I$ into a CNF formula T in DIMACS format, and builds an object representing a dictionary which makes a 1-1 correspondence between ground atoms of the Herbrand base of $S \cup I$ and propositional variables of the vocabulary. The file in DIMACS format is given as an input to a SAT solver (the choice of the SAT solver is completely independent of the application, as long as it accepts the standard DIMACS format as input, and can be chosen by the user), which delivers either a model of T, if satisfiable, or the indication that it is unsatisfiable. At this point, the Model2Spec module performs, using the dictionary, a backward translation of the model (if found) into the original language of the specification.

3 Benchmark Problems

As mentioned in Section 1, in this paper we show results in solving typical Constraint Programming benchmark problems using SAT technology, by taking advantage of Spec2Sat. In this section, we list the problems used for the experiments, that are a significant subset of those present in the well-known library CSPLib (`www.csplib.org`). We chose eight problems which cover five out of the seven classes of problems offered by the CSPLib. As a side-effect, we obtain declarative specifications (in the language NPSpec) for such problems. Due to lack of space, we describe (apart for Social golfer, presented in Section 2) the NPSpec specifications of only few of them. The others (and several more) can be found on-line.

Golomb rulers (problem nr. 006). A Golomb ruler of length L with m marks is defined as a set of m integers $0 = a_1 < a_2 < ... < a_m = L$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 \leq i < j \leq m$ are all distinct. In our formulation, given L and m, we are interested in finding a Golomb ruler of any length *not greater than L*. An NPSpec specification for this problem is as follows:

```
IntFunc({1..N_MARKS}, ruler, 0..LENGTH).                          // G1
fail <-- NOT ruler(1,0).                                          // G2
fail <-- ruler(I,V_I), ruler(J,V_J), J > I, V_I >= V_J.           // G3
fail <-- ruler(I,V_I), ruler(J,V_J), ruler(K,V_K), ruler(L,V_L), // G4
```

```
        I < J, K < L, I != K, V_J - V_I == V_L - V_K.
fail <-- ruler(I,V_I), ruler(J,V_J), ruler(K,V_K), ruler(L,V_L), // G5
        I < J, K < L, J != L, V_J - V_I == V_L - V_K.
```

The search space is defined (metarule G1) as the set of all total integer functions from {1..N_MARKS} to the integer range {0..LENGTH}, hence assigning a position on the ruler to each mark. Rule G2 forces the first mark to be at the beginning of the ruler (position 0). Rule G3 forces marks to be in ascending order, i.e., the second mark is on the right of the first one, the third on the right of the second, and so on. Rules G4 and G5 force distances between two marks to be all different.

All-interval series (problem nr. 007). Given the twelve standard pitch-classes (c, c#, d, ...), represented by numbers 0,1,...,11, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). Here, we generalize the problem by replacing the twelve standard pitch-classes with an arbitrary set pitch of N pitch-classes (all-interval series problem of size N). An ad-hoc encoding of this problem into SAT has been made in [6]. In NPSpec, the All-interval series problem can be specified as follows:

```
Permutation(pitch, series).                                        // A1
Permutation(interval, neighbor).                                   // A2
fail <-- series(P,X), series(Q, X+1), NOT neighbor(abs(P-Q), X). // A3
```

By metarule A1, guessed predicate series assigns a different order number to every pitch in the series. Metarule A2 guesses a second guessed predicate, neighbor, to be an ordering of all possible intervals (interval is defined in the instance file to be the integer range $[1, N-1]$): a tuple $\langle intv, idx\rangle$ in neighbor means that pitches at positions idx and $idx+1$ in the series are divided by interval $intv$. The final rule A3 actually forces series to respect this constraint: for each pair of adjacent pitches P and Q (at positions X and X+1), the interval dividing them must have order X in the permutation neighbor.

Schur's lemma (problem nr. 015). The problem is to put N_BALLS balls labelled {1,...,N_BALLS} into N_BOXES boxes so that for any triple of balls (x, y, z) with $x+y=z$, not all are in the same box. An NPSpec specification for this problem is as follows:

```
Partition({1..N_BALLS}, putIn, N_BOXES).                        // S1
fail <-- putIn(X,Box), putIn(Y,Box), putIn(Z,Box), X + Y == Z. // S2
```

Metarule S1 declares the search space to be the set of all partitions of the set of N_BALLS balls into N_BOXES boxes, while rule S2 expresses the constraint.

The other problems we considered are *Ramsey* (nr. 017) *Magic square* (nr. 019), *Langford's number* (nr. 024), and *Balanced academic curriculum* (BACP, nr. 030).

4 Experiments

For all problems we chose a non-trivial set of instances – by using, when possible, publicly available benchmarks (e.g. for BACP) – and compiled them into SAT. Then, we ran different SAT solvers on those instances, and measured their solving times. We used two recent SAT solvers, very different in nature: *(i)* zChaff, one of the fastest, complete solvers today available and *(ii)* bg-walksat, a sound but incomplete one, based on local search. bg-walksat is a recent extension of the well-known walksat, where the search is guided by *backbones* of the formula. Furthermore, we investigated whether the application of different reformulation techniques was suitable for improving solvers' performances. In particular, we applied two, in some sense, complementary techniques: adding symmetry-breaking constraints and neglecting *safe-delay* constraints.

Symmetry-breaking. The presence of symmetries in CSPs has been widely recognized to be one of the major obstacles for their efficient resolution. To this end, different approaches have been followed in the literature in order to deal with them, the best known being that of adding proper –i.e., symmetry-breaking– constraints to the CSP model (cf., e.g., [13,4]). Along the lines of [9], we added symmetry-breaking constraints at the specification level, hence breaking "structural" symmetries (i.e. those that depend on the problem structure, and not on the particular instance considered). Such approach has been proved to be effective for different classes of solvers, on different problems, and comes natural when using a purely declarative modelling language.

Safe-delay constraints. Given a specification, a safe-delay constraint is a constraint whose evaluation can be safely ignored in a first step, hence simplifying the problem, and efficiently reinforced in a second step, when a solution to the relaxed problem has been found [2]. The importance of safe-delay constraints is that their reinforcement can always be done in polynomial time, without further search. Highlighting and delaying safe-delay constraints can be very effective because: *(i)* The set of solutions is enlarged, and this can be beneficial for some solvers; *(ii)* The instantiation can be more efficient, since fewer constraints have to be grounded: this of course applies also to the SAT case, where delaying constraints reduces the number of generated clauses; *(iii)* The reinforcement of delayed constraints is often very efficient, e.g., linear or logarithmic time in the size of the input (cf. examples below). It is worth noting that also the deletion of safe-delay constraints is done by reformulating the declarative specification of the problem, hence independently of the instance.

Right now, all the reformulations have been performed manually. However, in previous work [2,9,1], we showed how the required forms of reasoning can be in principle autonomously made by system, since they reduce to check properties of first-order formulae.

For the problems presented in Section 3, we considered the following instances:

- Golomb rulers: lengths up to 15, and numbers of marks up to 9;
- All-interval: pitch classes up to 18 (zChaff) and to 40 (bg-walksat);

Table 1. Results of the experiments using zChaff (a) and bg-walksat (b)

Problem name	zChaff					
	Instances			SAT compil	SAT solving	Total
	nr.	solved	unsolved	time (sec)	time (sec)	time (sec)
Golomb Ruler	34	34	0	39412.96	2.46	39415.42
with symm breaking	34	34	0	40366.67	2.82	40369.49
with safe delay	34	34	0	26654.29	27.66	26681.95
All-Interval Series	14	13	1	6.29	6600.70	6606.99
with symm breaking	14	10	4	6.55	17265.64	17272.19
Social Golfer	168	110	58	66171.70	212527.78	278699.48
with symm breaking	168	162	6	62774.72	25185.60	87960.32
Schur's Lemma	164	164	0	2412.57	0.08	2412.65
with safe delay	164	164	0	2510.13	0.12	2510.12
with symm breaking	164	164	0	2537.14	0.08	2537.22
Ramsey problem	85	82	3	155.24	10803.04	10958.28
with safe delay	85	82	3	153.95	10802.61	10956.56
with symm breaking	85	82	3	154.64	10802.49	10957.13
Magic Square	3	3	0	281.16	128.59	409.75
with symm breaking	3	3	0	282.03	38.25	320.28
Langford's number	43	39	4	1982.14	18109.22	20091.36
with symm breaking	43	39	4	1983.91	17422.39	19406.3
BACP	2	2	0	2041.13	1.11	2042.24

(a)

Problem name	bg-walksat				
	Instances		SAT compil	SAT solving	Total
	nr.	success ratio	time (sec)	time (sec)	time (sec)
Golomb Ruler	20	100%	15274.17	3528.55	18802.72
with symm breaking	20	100%	16285.75	4632.28	20918.03
with safe delay	20	60%	7617.11	6315.08	13932.19
All-Interval Series	36	17%	171.21	702.98	874.19
with symm breaking	36	14%	172.51	689.2	861.68
Social Golfer	137	43%	16453.92	3633.48	20087.40
with symm breaking	137	46%	17132.50	3792.73	20925.23
Schur's Lemma	164	100%	2412.57	4.32	2416.89
with safe delay	164	99%	2510.00	4.18	2514.18
with symm breaking	164	100%	2537.14	7.03	2544.17
Ramsey problem	85	94%	155.24	8.47	163.71
with safe delay	85	100%	153.95	7.49	161.44
with symm breaking	85	94%	154.64	8.05	162.69
Magic Square	3	33%	281.16	31.97	313.13
with symm breaking	3	33%	282.03	32.07	314.10
Langford's number	33	76%	549.76	355.53	905.29
with symm breaking	33	67%	549.74	357.63	907.37

(b)

– Social golfer: up to 36/3/6, or up to 25/6/5 (players/weeks/groups);
– Schur's lemma: up to 50 balls and 10 boxes;
– Ramsey problem: up to 19 nodes and 7 colors;
– Magic squares: sizes up to 4;
– Langford's number: up to 4 sets and 14 numbers;
– BACP: 2 benchmark instances, taken from CSPLib, solved with zChaff.

Results of our experiments are shown in Table 1, where (a) and (b) refer to zChaff and bg-walksat, respectively, and list the overall times for compiling into SAT and solving the whole set of instances for each problem. Column "Total time" gives the gross time needed for processing the whole set of instances, summing up times for compilation and solving. Both stages had a timeout of

1 hour per instance. Finally, column "nr." contains the number of instances run (for some problems, we were unable to solve the largest instances with BG-WALKSAT, obtaining a *too-many-clauses* error: for this reason, in Table 1(b) the number of instances is sometimes smaller), column "solved" ("unsolved") shows how many instances have been successfully (unsuccessfully) processed within the timeout, either if sat or unsat (unsolved instances contribute with 3600 secs to the total time); since BG-WALKSAT is an incomplete solver, column "success ratio" (in (b)) reports the percentage of instances for which the correct answer is given. In addition, rows "with safe delay" and "with symmetry breaking" show the behavior of the two solvers when performing reformulation. In particular, we reformulated the following problems by safe-delay:

– Golomb rulers: rule `G3` can be ignored, enlarging the set of solutions by all their permutations. However, in this case a simple modification of the other constraints is required [2]: in particular, the *absolute values* of distances between marks have to be different.
– Schur's lemma: we let balls to be put in more than one box at the same time. If such a solution exists, a valid solution of the original problem can be derived by arbitrarily choosing a single box for each ball.
– Ramsey problem: we let multiple colors to be assigned to the same node. If such a coloring exists, then it suffices to arbitrarily choose an arbitrary color for each node having multiple ones.

In the last two cases, ignoring safe-delay constraints yields to guess multi-valued functions for the guessed predicates. This task can be accomplished by the current implementation of SPEC2SAT by defining a guessed predicate as a `multivalued` partition or integer function. We also observe that for all three problems, the second stage, i.e., recovering a solution to the original problem from a solution to the simplified one, can be performed very efficiently: in $m \log m$ for Golomb rulers (by ordering marks), and in linear time for both Ramsey and Schur's lemma.

Concerning symmetry-breaking, we broke some of the symmetries in all but one problems. We proceeded as follows: in Golomb rulers we forced the distance between the first two marks to be less than that between the last two; in All-interval and Langford problems we forced, respectively, the first pitch-class and the first number to be less than the last one. As for Social golfer, we fixed the scheduling for the first and partially the second week, and assigned the first player always to the first group. Finally, as for Ramsey, Schur's lemma and Magic square, we fixed the choice for the first ball/edge/square, respectively.

Some comments on the results in Table 1 are in order. First of all, both SAT solvers behave well in many cases, being able to solve instances of reasonable size. However, it is not the case that one solver is always better than the other: ZCHAFF seems much faster than BG-WALKSAT in solving Golomb rulers or BACP instances (BG-WALKSAT was not able to run on instances of the latter problem, due to their large size), while the latter performs better on All-intervals series, Ramsey, and Social golfer, even if its success ratio is not always very high. Interestingly, applying the two reformulation techniques sometimes greatly helps zChaff. As an example, by ignoring safe-delay constraints on Golomb Rulers, the overall compilation time falls down of about 13000 seconds, while the solving time increases only of about 25 seconds. A similar behavior happens also when solving this problem with bg-walksat.

zChaff is also positively affected by symmetry-breaking. As for Magic square and Social golfer, the speed-up is impressive. It is interesting to observe that the compilation times do not grow significantly, since the symmetry-breaking constraints we chose are quite simple. On the other hand, Schur's Lemma and Ramsey do not show any improvement. Intuitively, this is due to the lack of unsatisfiable instances in the considered set. In fact, it is well known that symmetry-breaking produces its best effects on unsatisfiable instances, avoiding the search engine to explore the whole search space before terminating with a negative answer. Unfortunately, for Schur's lemma and Ramsey, no negative instances have been found that could be handled within the timeout. Another interesting aspect is that the local search solver bg-walksat does not take benefit from symmetry-breaking, hence confirming the intuition that a reduction of the solution density is an obstacle for local search [12,9].

5 Discussion and Conclusions

In this paper, we discussed how the availability of specification languages for constraint problems that automatically compile instances into SAT, can make SAT solving technology an effective tool for Constraint Programing. We experienced NPSpec and Spec2Sat on several well-known benchmark problems for CP, and demonstrated how they can be easily formulated in NPSpec, and solved by exploiting state-of-the-art SAT solvers. Additionally, we showed how applying reformulation techniques such as ignoring safe-delay or adding symmetry-breaking constraints to the specifications can be very effective in improving solvers' performances. Experiments also show that the most critical point when using SAT as an engine for solving constraint problems is the compilation task, which can be, in some cases, very expensive (cf. Table 1). However, often this is compensated by the much higher (and continuously improving) efficiency of SAT solvers wrt, e.g., CP ones. In particular, experiments that have been carried out involving the state-of-the-art CP solver opl [15] (a detailed discussion will appear in the full paper) show that:

– For some problems, e.g., Golomb rulers, Schur's lemma, Magic square, CP is much faster (e.g., as for Schur's lemma, the overall time needed by opl is 52 seconds, against 2412 seconds needed by Spec2Sat+zChaff). However, in these cases, the real bottleneck is actually the compilation step (e.g., the actual time needed by zChaff to solve all instances of such problem is only 0.08 seconds).
– For others, e.g., All-Interval, Langford's number, and Ramsey, the time needed by opl is higher than that needed by Spec2Sat+zChaff (8328 vs. 6606, 22100 vs. 20091, 18030 vs. 10958 secs, respectively). Interestingly, the opl models for the first two problems exhibit global constraints (*all-different*

and *distribute*, respectively), that are one of the most important features of CP solvers wrt SAT, and that may greatly improve their performance.

In our opinion, such behavior suggests that SAT can undoubtably be considered a respectable candidate for competing with CP on combinatorial problems, and that research on more efficient techniques for compiling into SAT is a real challenge. The existence of reformulation techniques, like safe-delay, that can have a positive impact on compilation times, and recent results in related fields like Answer Set Programming suggest promising lines of research.

References

1. M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. In *Proc. of AI*IA 2005*, v. 3673 of *LNAI*, pp. 38–49, 2005.
2. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. *Artif. Intell.*, 2006. To appear.
3. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162:89–120, 2005.
4. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of KR'96*, pp. 148–159, 1996. Morgan Kaufmann.
5. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming.* Intl. Thomson Publ., 1993.
6. H. H. Hoos. *Stochastic Local Search - Methods, Models, Applications.* PhD thesis, TU Darmstadt, 1998.
7. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI'96*, pp. 1194–1201, 1996. AAAI Press/The MIT Press.
8. 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.
9. T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proc. of SARA 2005*, v. 3607 of *LNAI*, pp. 165–181, 2005. Springer.
10. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000.
11. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC 2001*, pp. 530–535, 2001. ACM Press.
12. S. D. Prestwich and A. Roli. Symmetry breaking and local search spaces. In *Proc. of CPAIOR 2005*, v. 3524 of *LNCS*, pp. 273–287, 2005. Springer.
13. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proc. of ISMIS'93*, v. 689 of *LNCS*, pp. 350–361, 1993. Springer.
14. B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proc. of DIMACS Challenge on Cliques, Coloring, and Sat.*, 1993.
15. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.
16. W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proc. of IJCAI 2003*, pp. 1179–1186, 2003. Morgan Kaufmann.