SAT as an effective solving technology for constraint problems

Preliminary report

Marco Cadoli, Toni Mancini, 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 make a preliminary investigatation on 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 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 preliminary but 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 (cf., e.g., [21, 15]), 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., [28]), mathematical programming (cf., e.g., [13]), answer sets and stable model semantics (cf., e.g., [10, 20]), which are typically complete, or those that rely on local-search techniques (cf., e.g., [18, 29]) 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 has 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 [19], and incomplete ones, such as walksat [25],

and BG-WALKSAT [30]. 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 has led a great interest in the CP research community, and many papers show how to translate (compile) into SAT instances of various problems, like, e.g., scheduling, planning, or combinatorial ones (cf., e.g., [8, 17, 11, 16]). 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 that compile problem instances into SAT formulae, e.g., the language NPSPEC and the SPEC2SAT system [2, 7], 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 from the instances.

In this paper, we present some preliminary experiments showing that SAT technology can be *effectively* used 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 [15], 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. For the future, we plan to strengthen our evaluation by adding more problems from CSPLib, run the same problems using other systems, including OPL [28] smodels [20], and dlv [10], and possibly adopting more SAT solvers.

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). Such techniques include, e.g., adding new constraints to the model, such as symmetry-breaking ones, or the somewhat opposite strategy of ignoring some of the constraints that are guaranteed to be reinforced in a later stage (the so called *safe-delay constraints* [4]). We started experiencing the application of the above techniques while performing our experiments, and present some results. It is worth noting that these techniques are applied at the symbolic level of the specification, and hence independently on the instance. This possibility further increases the level of declarativeness of the modelling stage, which is fundamental in order to effectively take advantage of SAT technology.

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 concludes the paper.

2 The NPSpec language and Spec2Sat

NPSpec and Spec2Sat have been extensively described in [7]. 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.

2.1 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 [1], which can include the six predefined relational operators and negative literals.

The full syntax of NPSpec is given in [7], hence here we just recall it with an example. In particular, we show an NPSpec program for the *Hamiltonian path* NP-complete problem [14, Prob. GT39, p. 199], i.e., the problem where the input is a graph and the question is whether a traversal exists that touches each node exactly once.

```
DATABASE
  n = 6;  // no. of nodes
  edge = {(1,2),(3,1),(2,3),(6,2),(5,6), (4,5),(3,5),(1,4),(4,1)};
SPECIFICATION
  Permutation({1..n},path).                          // H1
  fail <-- path(X,P), path(Y,P+1), NOT edge(X,Y). // H2
```

The following comments are in order:

- The input graph is defined in the `DATABASE` section, which is generally provided in a separate file.
- In the search space declaration (metarule `H1`) the user declares the predicate symbol `path` to be a "guessed" one, implicitly of arity 2. 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.
- `path` is declared to be a permutation of the finite domain {`1..n`}. This means that its extension must represent a permutation of order 6. As an example, $\{(1,5),(2,3),(3,6),(4,2),(5,1),(6,4)\}$ is a valid extension.
- Comments can be inserted using the symbol "`//`".
- Rule `H2` is the constraint that permutations must obey in order to be Hamiltonian paths: a permutation `fail`s, i.e., it is not valid, if two nodes `X` and `Y` which are adjacent in the permutation are not connected by an edge. `X` and `Y` are adjacent because they hold places `P` and `P+1` of the permutation, respectively.

Running this program on the NPSpec compiler produces the following output:

```
path: (1, 1) (2, 5) (3, 6) (4, 2) (5, 3) (6, 4)
```

which means "1 is the first node in the path, 4 is the second node in the path, ..., 3 is the sixth node in the path", and is indeed an Hamiltonian path.

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:

1. `Subset(<domain>, <pred_id>).`
2. `Permutation(<domain>, <pred_id>).`
3. `Partition(<domain>, <pred_id>, n).`
4. `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. As an example, the program for the Hamiltonian path can be extended to the Hamiltonian cycle problem [14, Prob. GT37, p. 199] by adding the following rule

```
fail <-- path(X,n), path(Y,1), NOT edge(X,Y).     // H3
```

Moreover, undirected graphs can be handled by including a further literal `NOT edge(Y,X)` in the body of both rules `H2` and `H3`.

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.

2.2 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 [7], but just briefly describe the general architecture of the application, shown in Figure 1.

The module Parser receives text files containing the specification S in NPSpec and the instance data I, parses them, and builds its internal representation. The module 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 from 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

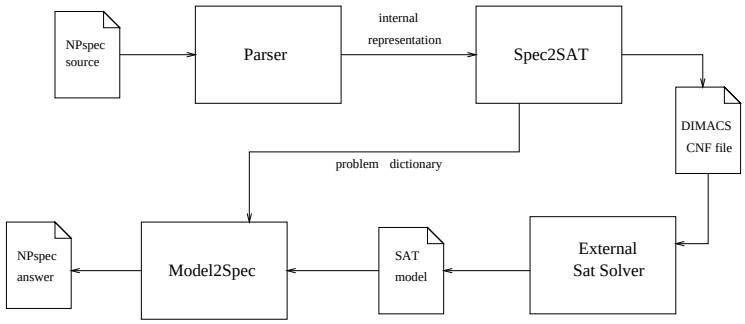

Fig. 1. Architecture of the NPSpec compilation and execution environment.

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. Appropriate interfaces for different SAT solvers and the Model2Spec module have, of course, been written, in order to translate the (proprietary) format for the output, into Model2Spec.

3 Benchmark problems

As mentioned in Section 1, in this paper we show our preliminary experiments in solving typical Constraint Programming benchmark problems using SAT technology, by taking advantage of NPSpec. In this section, we list the problems used for our experiments, that are a significant subset of those present in the well-known library CSPLib [15]. For the experiments of this paper we chose seven problems which cover five out of the seven classes of problems offered by the CSPLib. More specifications can be found on-line. As a side-effect, we obtain declarative specifications (in the language NPSpec) for such problems. The number in parentheses close to each problem is its identification number in the library:

Golomb ruler (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,        // G3
         V_I >= V_J.
fail <-- ruler(I,V_I), ruler(J,V_J), ruler(K,V_K), // G4
```

```
        ruler(L,V_L), 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), // G5
        ruler(L,V_L), 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 [16]. 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), // A3
         NOT neighbor(abs(P - Q), X).
```

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`.

Social golfer (problem nr. 010). Given `N_PLAYERS` = `N_GROUPS` $*$ `GROUP_SIZE` golf players, this problem amounts to find an arrangement for all of them into `N_GROUPS` groups of size `GROUP_SIZE` over `N_WEEKS` weeks, in such a way that no player plays in the same group as any other in more than one week.

```
IntFunc({1..N_PLAYERS}><{1..N_WEEKS}, play, 1..N_GROUPS).

fail <-- play(P1,W1,G1), play(P2,W1,G1), P1 != P2,
         play(P1,W2,G2), play(P2,W2,G2), W1 != W2.
fail <-- COUNT(play(*,W,G),X), X != GROUP_SIZE.
```

Schur's lemma (problem nr. 015). The problem is to put `N_BALLS` balls labelled $\{1, ..., \texttt{N\_BALLS}\}$ 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), // S2
         X + Y == Z.
```

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.

Ramsey problem (problem nr. 017). The Ramsey problem is to color the edges of a complete graph with n nodes using at most k colors, in such a way that there is no monochromatic triangle in the graph. A specification is as follows:

```
Partition({1..N_EDGES}, coloring, N_COLORS).     // R1

fail <-- edge(X,A,B), edge(Y,B,C), edge(Z,A,C),  // R2
         coloring(X,Col), coloring(Y,Col),
         coloring(Z,Col).
```

Magic square (problem nr. 019). A magic square of order N is an N by N matrix containing the numbers 1 to N^2, with numbers on each row, column and main diagonal having the same sum.

An NPSpec specification for this problem is as follows:

```
  Permutation({1..N}><{1..N}, square).                              // M1
  Permutation({1..N^2}, magic_square).                              // M2

  Subset({1..N^2}, diag).                                           // M3
  Partition({1..N^2}, column, N).                                   // M4
  Partition({1..N^2}, row, N).                                      // M5

  // Channeling constraints for "square"
  square(X,Y,I) <-- I == (X -1)*N + Y.                              // M6
  fail <-- square(X,Y,I), I != (X -1)*N + Y.                        // M7

  // Channeling constraints for "diag"
  fail <-- NOT diag(X), magic_square(X,I), square(R,C,I), R == C. // M8
  fail <-- diag(X), magic_square(X,I), square(R,C,I), R != C.     // M9

  // Channeling constraints for "column"
  fail <-- NOT column(V,C-1), magic_square(V,I), square(_,C,I).   // M10

  // Channeling constraints for "row"
  fail <-- NOT row(V,R-1), magic_square(V,I), square(R,_,I).      // M11
```

```
fail <-- SUM(column(*,I),C:0..MAX_SUM),                          // M12
         SUM(diag(*),D:0..MAX_SUM), C != D.
fail <-- SUM(row(*,I),C:0..MAX_SUM),                             // M13
         SUM(diag(*),D:0..MAX_SUM), C != D.
```

Metarule M1, together with rules M6 and M7, defines square to be an enumeration of all the entries of the magic square (each of them having coordinates in $[1, N] \times [1, N]$). Hence, each entry is assigned an index in $[1, N^2]$. Metarule M2 then guesses a value in $[1, N^2]$ for each entry. Such values are all different.

The other guessed predicates (rules M3, ..., M5) are redundant, and represent a more convenient representation of the values in the main diagonal, in each row and column respectively. Rules M8, ..., M11 define them starting from magic_square, acting as channeling constraints.

Finally, rules M12 and M13 force entries to be such that the sum of values in the main diagonal, every row and every column are all equal. MAX_SUM is an instance-dependent value representing the maximum value for the sums.

Langford's number (problem nr. 024). The $L(k, n)$ problem is to arrange k sets of numbers 1 to n, so that each appearance of the number m is m numbers on from the last. As an example, the $L(3, 9)$ problem is to arrange 3 sets of the numbers 1 to 9 so that the first two 1's and the second two 1's appear one number apart, the first two 2's and the second two 2's appear two numbers apart, etc. A specification is as follows:

```
IntFunc({1..NUMBERS*SETS}, sequence, 1..NUMBERS). // L1

fail <-- COUNT(sequence(*,_),X), X != SETS.       // L2
fail <-- sequence(I,N), NOT sequence(I+N+1,N),    // L3
         sequence(J,N), J > I.
```

Metarule L1 declares the guessed predicate sequence to be a function from {1..NUMBERS*SETS} to 1..NUMBERS. Rule L2 forces each number to appear exactly SETS times in the sequence, while L3 checks whether the distances between two occurrences of the same number are correct.

Balanced academic curriculum problem (problem nr. 030). The Balanced academic curriculum problem (BACP) amounts to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible. The curriculum must obey several administrative and academic regulations, such that the respect of prerequisites among courses, minimum and maximum number of courses per periods, as well as minimum and maximum academic load in each period.

```
Partition(load, curriculum, PERIODS).

fail <-- SUM(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,M,_,_,_), X < M.
fail <-- SUM(curriculum(?,*,P), X:0..MAX_LOAD),
```

```
         period(P,_,M,_,_), X > M.
fail <-- COUNT(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,_,_,M,_), X < M.
fail <-- COUNT(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,_,_,_,M), X > M.
fail <-- prerequisite(Pre, Post), curriculum(Pre,_,P),
         curriculum(Post,_,Q), Q <= P.
```

4 Experiments

We chose a non-trivial set of instances for each problem defined above, by using publicly available benchmarks when possible (e.g., for BACP), and compiled all such instances into SAT ones. Then, we ran different SAT solvers on those instances, and measured their solving times. In this preliminary stage, we used two recent SAT solvers, very different in nature:

- zChaff, one of the fastest, complete solvers today available;
- 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, as already mentioned in Section 1, we started investigating whether the application of different reformulation techniques was suitable for improving solvers' performances. In particular, we applied two different, and in some sense, complementary, techniques: adding symmetry-breaking constraints and neglecting *safe-delay* constraints. These techniques are briefly described in the following:

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 most well known one is to add additional constraints to the CSP model, that filter out many (hopefully all but one) of the symmetrical points in the search space. These are called symmetry-breaking constraints, cf., e.g., [23, 9, 24, 26, 27, 12].

We used this approach with a major difference, adding symmetry-breaking constraints at the level of the problem specification. Hence, we broke "structural" symmetries of the problems, i.e., those symmetries that depend on the problem structure, and not on the particular instance considered. Breaking symmetries at the specification level has been proved to be effective for different classes of solvers, on different problems [3], and comes natural when using a purely declarative modelling language such as NPSpec.

Safe-delay constraints. Given a problem 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 [4]. 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 useful when solving constraint problems, at least for three reasons: *(i)* For every instance, the set of solutions is enlarged (since some constraints are removed), and this can be beneficial for some classes of solvers. *(ii)* The instantiation stage can be done more efficiently, since a fewer number of constraints have to be grounded: this is the case also when using SAT technology, since delaying constraints reduces the number of clauses generated during instantiation. *(iii)* The reinforcement of delayed constraints (which is guaranteed to be polynomial time) is often very efficient, e.g., linear of logarithmic time in the size of the input, (we show some examples in Section 4). It is worth noting that also the deletion of safe-delay constraints is done by reformulating the declarative specification of the problem, hence independently on the instance.

For the various problems of Section 3, we used the following instances:

- Golomb ruler: lengths up to 15, with up to 9 marks when using zChaff, and lengths up to 12, with up to 5 marks when using bg-walksat;
- All-interval series: pitch classes up to 18 (zChaff) and up to 40 (bg-walksat);
- Social golfer: up to 8 players, 6 weeks;
- Schur's lemma: up to 50 balls and 10 boxes;
- Ramsey problem: up to 20 nodes and 7 colors (zChaff), and 12 nodes and 5 colors (bg-walksat);
- Magic squares: sizes up to 5, when using zChaff, and 4 when using bg-walksat;
- Langford's number: up to 4 sets and 19 numbers (zChaff), and 4 sets and 10 numbers (bg-walksat);
- BACP: 2 benchmark instances, taken from CSPLib, solved with zChaff.

Results of our experiments are shown in Table 1 (both compilation and solving processes had a timeout of 1 hour). In particular, Table 1(a) shows compilation and solving times for zChaff, while Table 1(b) shows times when using bg-walksat on the same problems.

For each solver and problem, we report the number of instances run, and the number of those solved successfully in 1 hour. As for the incomplete bg-walksat instead, we report its success ratio, i.e., the percentage of the instances for which this solver gave the correct answer. Then, we list the overall times for compiling and solving the whole set of instances for each problem.

Moreover, we investigate the effectiveness of the reformulation techniques discussed above. In particular, we ignored safe-delay constraints on the following problems:

- Golomb ruler: rule `G3` of the problem specification can be ignored, hence enlarging the set of solutions by all their permutations. However, in this case, a simple modification of the other constraints is required [4]): in particular, the *absolute values* of distances between marks have to be different.

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 safe delay	34	34	0	26654.29	27.66	26681.95
All-Interval Series	14	13	1	6.29	6600.70	6606.99
Social Golfer	168	110	58	64467.93	212527.78	276995.71
with symm breaking	168	162	6	62774.72	3782.73	66567.45
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	9099.76	484.017	9583.78
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	41	2	1982.14	18109.22	20091.36
BACP	2	2	0	2798.85	2.20	2801.05

(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 safe delay	20	60%	7617.11	6315.08	13932.19
All-Interval Series	36	17%	171.21	702.98	874.19
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	36	67%	813.64	355.53	1169.17

(b)

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

- 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.

We note that, in the last two cases, ignoring safe-delay constraints reduces 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 ruler (by ordering marks), and in linear time for both

Ramsey and Schur's lemma problems (we remind that the reinforcement of safe-delay constraints is always guaranteed to be polynomial).

As for symmetry-breaking instead, we broke some of the symmetries in the Social golfer, Schur's lemma, Magic square, and Ramsey problems. In particular, as for Social golfer, we fixed the scheduling for the first two weeks, and made the first player play always in the first group (these symmetry-breaking constraints are shown in [27]). As for Schur's lemma and Magic square, we simply fixed the choice for the first edge and the first square, respectively. As for Ramsey instead, a slighly more complex symmetry-breaking constraint is added, by fixing a suitable ordering on the colors and on the edges.

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 for solving Golomb ruler or BACP instances (bg-walksat was not able to run on instances of the latter problem, due to their large size), while the latter is better for All-intervals series, Ramsey, and Social golfer, even if its success ratio (i.e., the ratio of satisfiable instances for which this incomplete solver was actually able to find a solution) 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 Ruler, 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, Ramsey, 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 (apart for Ramsey, where a more complex symmetry-breaking constraint was used). It is worth noting that adding more complex constraints to these problems led to poorer performances for both Spec2Sat and zChaff. 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 (cf., e.g., [22, 3]).

5 Conclusions and future work

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 constraints or adding

symmetry-breaking constraints to the problem specifications can be very effective in improving solvers' performances, and/or easing the compilation task.

It is worth noting that such techniques have already been used in ad-hoc SAT encodings of particular CP problems. As an example, the standard DIMACS SAT encoding of Graph k-coloring, actually omits the "at-most-one color" constraint of the problem, that is actually safe-delay [4]. The same happens in the SAT encoding for Job shop scheduling given in [8], where propositional variables represent the encoding of *earliest starting times* and *latest ending times* for all tasks, rather than their exact scheduled times. As for symmetry-breaking, ad-hoc SAT encodings of various problems usually take care of breaking (some of the) symmetries in the generated SAT instances.

The main difference of our work with respect to the others, is that the user can perform such optimizations at the symbolic level of the specification, by relying on a purely declarative language such as NPSpec. Right now, this is done manually by the NPSpec modeller, however, in some previous work, we showed how recognizing whether a reformulation technique can be applied, can be in principle autonomously made by system, since it reduces to check properties of first-order logic formulae [4, 3, 6]. Of course, since the goodness of a particular optimization technique is expected to depend both on the problem and on the solver (cf. results in Table 1), a much wider experimentation is needed, using a larger number of solvers on a wider set of problems. A deeper investigation of the applicability of other techniques to the SAT case (cf., e.g., [5]) is also planned.

Another important point that lacks in the current preliminary version of this work is the comparison of SAT with respect to CP solvers for the investigated problems. To this end, we plan to make a much wider experimentation that involves also state-of-the-art CP solvers, e.g., among the others, the well known OPL language [28], by Ilog.

References

1. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, Los Altos, 1988.
2. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.
3. M. Cadoli and T. Mancini. Detecting and breaking symmetries on specifications. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, Kinsale, Ireland, 2003.
4. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 388–398, Whistler, BC, Canada, 2004. AAAI Press/The MIT Press.
5. M. Cadoli and T. Mancini. Exploiting functional dependencies in declarative problem specifications. In *Proceedings of the Ninth European Conference on Logics in*

Artificial Intelligence (JELIA 2004), volume 3229 of *Lecture Notes in Artificial Intelligence*, Lisbon, Portugal, 2004. Springer.

6. M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. In *Proceedings of the Third International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Toronto, Canada, 2004.
7. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.
8. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097, Seattle, WA, USA, 1994. AAAI Press/The MIT Press.
9. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, Cambridge, MA, USA, 1996. Morgan Kaufmann, Los Altos.
10. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417, Trento, Italy, 1998. Morgan Kaufmann, Los Altos.
11. M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 1169–1177, Nagoya, Japan, 1997. Morgan Kaufmann, Los Altos.
12. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, page 462 ff., Ithaca, NY, USA, 2002. Springer.
13. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, USA, 1979.
15. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, APES-09-1999, 1999. Available from `http://www.csplib.org`.
16. H. H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, TU Darmstadt, 1998.
17. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201, Portland, OR, USA, 1996. AAAI Press/The MIT Press.
18. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000.
19. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Thirtyeighth Conference on Design Automation (DAC 2001)*, pages 530–535, Las Vegas, NV, USA, 2001. ACM Press.
20. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

21. OR Library. Available at `www.ms.ic.ac.uk/info.html`.
22. S. Prestwich. Supersymmetric modeling for local search. In *Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, Ithaca, NY, USA, 2002.
23. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In H. J. Komorowski and Z. W. Ras, editors, *Proceedings of the Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93)*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361, Trondheim, Norway, 1993. Springer.
24. E. Rothberg. Using cuts to remove symmetry. In *Proceedings of the Seventeenth International Symposium on Mathematical Programming (ISMP 2000)*, Atlanta, GA, USA, 2000.
25. B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In M. Trick and D. S. Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence, RI, USA, 1993.
26. H. D. Sherali and J. Cole Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 47:1396–1407, 2001.
27. B. Smith. Reducing symmetry in a combinatorial design problem. In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2001)*, pages 351–360, Ashford, Kent, UK, 2001.
28. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
29. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*, pages 65–80, Kinsale, Ireland, 2003. Springer.
30. W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1179–1186, Acapulco, Mexico, 2003. Morgan Kaufmann, Los Altos.