Compiling Problem Specifications into SAT

Marco Cadoli[1] and Andrea Schaerf[2]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy
cadoli@dis.uniroma1.it http://www.dis.uniroma1.it/~cadoli
[2] Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica
Università di Udine, Via delle Scienze 208, I-33100 Udine, Italy
schaerf@uniud.it http://www.diegm.uniud.it/schaerf

Abstract. We present a compiler that translates a problem specification into a propositional satisfiability test (SAT). Problems are specified in a logic-based language, called NP-SPEC, which allows the definition of complex problems in a highly declarative way, and whose expressive power is such to capture exactly all problems which belong to the complexity class NP. The target SAT instance is solved using any of the various state-of-the-art solvers available from the community. The system obtained is an executable specification language for all NP problems which shows interesting computational properties. The performances of the system have been tested on a few classical problems, namely graph coloring, Hamiltonian cycle, and job-shop scheduling.

1 Introduction

We present a system for writing and executing specifications for search problems, which makes use of NP-SPEC, a highly declarative specification language. NP-SPEC has a DATALOG-like, i.e., PROLOG with no function symbols, syntax; its semantics is based on the notion of *model minimality*, an extension of the well-known least-fixed-point semantics of the Horn fragment of first-order logic [26]. NP-SPEC allows the user to express every problem belonging to the complexity class NP [12], which notoriously includes many problems interesting for real-world applications. Restriction of expressiveness to NP guarantees termination and helps to obtain efficient executions.

The core of our system is the compiler, called SPEC2SAT, that translates problem specifications written in NP-SPEC into instances of the *propositional satisfiability* problem (SAT). An instance π of the original problem is translated into a formula T of propositional logic in conjunctive normal form, in such a way that T is satisfiable if and only if π has a solution. Moreover, from the variable assignments that satisfy T the system reconstructs the solution of π.

A specification S of π is a set of metarules defining the search space, plus a set of rules defining the admissibility function. Both metarules and rules are transformed into a set of clauses of T encoding their semantics. The translation of rules is based on their ground instantiation over the *Herbrand universe*. Our algorithm for instantiation uses complex auxiliary data structures so as to try as much as possible to avoid the generation of useless clauses.

The approach of translation into SAT is motivated by the huge amount of research devoted to such a problem in last years (see, e.g., [15]), and the number of fast solvers available from the research community. Such solvers, both complete and incomplete ones, are able to solve in a few seconds instances of hundreds of thousands of clauses; and this result was unconceivable only a few years ago. In addition, the community working on SAT is still very active, and even better SAT solvers can be expected to come up in the future.

SAT is the prototypical NP-complete problem, and every instance π of a problem in NP can be translated into an instance of SAT of polynomial size in the size of π. In practice, this idea has been exploited since several years for various problems such as planning [17,16], scheduling [7], theorem proving in finite algebra [11], generation of test patterns for combinatorial circuits [18], and cryptography [21]. Those papers showed that translating a problem into SAT can give good performance of the resulting system, when compared with state-of-the-art dedicated solvers.

The shortcoming of those previous works is that the translation had to be done completely by hand for each problem. Conversely, we aim at a system that automatically translates any NP problem into SAT using the simple and declarative language NP-SPEC.

In terms of performances NP-SPEC cannot compete with state-of-the-art solvers of well-studied problems, anyway we believe that it is a valuable tool for developing fast prototypes for new problems, or variations of known ones, for which no specific solver is available. Nevertheless, experimental results show that our system is able to solve in reasonable time medium-size instances of various classical problems. In addition, it works much faster that the original engine of NP-SPEC [5,3] which is based on a translation of the input specification in the logic programming language PROLOG.

2 Preliminaries

2.1 Overview of the NP-SPEC Language

An NP-SPEC program consists of a `DATABASE` section and a `SPECIFICATION` section. The former section includes the definition of extensional relations, and of integer intervals and constants. The latter section consists of two parts: a *search space* declaration, and a stratified DATALOG program [2], which can include the six predefined relational operators and negative literals.

As a first example, we show an NP-SPEC program for the *Hamiltonian path* NP-complete problem, i.e., the problem where the input is a graph and the question is whether there is a traversal that touches each node exactly once.

```
DATABASE
      NODES = 6;
      EDGE = {(1,2),(3,1),(2,3),(6,2),(5,6),(4,5),(3,5),(1,4),(4,1)};
```

```
SPECIFICATION
      Permutation({1..NODES},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.
– In the search space declaration (metarule H1) the user declares the predicate symbol path to be a "guessed" one 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..NODES}. 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 permutations must obey in order to be Hamiltonian paths: a permutation fails, 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 occupy places P and P+1 of the permutation, respectively.

Running this program on the NP-SPEC 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.

As another example, in the *graph coloring* NP-complete problem the input are a graph G and a positive integer k representing the number of available colors, and the question is whether it is possible to give each node of G a color in such a way that adjacent nodes are never colored the same way. The intuitive structure of the search space in this case is a *partition* of the nodes of G into k distinct subsets, since an assignment of nodes to colors must be guessed. The NP-SPEC program for checking colorability is:

```
DATABASE
      K = 3;
      N = 6;
      EDGE = {(1,2),(3,1),(2,3),(6,2),(5,6),(4,5),(3,5)};
SPECIFICATION
      Partition({1..N},coloring,K).                                // GC1
      fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).            // GC2
```

Another typical structure of the search space is the *integer function*, i.e., the assignment of a value in a specified domain to a set of variables. As an example, in the quadratic Diophantine equations NP-complete problem the input are three positive integers a, b, c, and the question is whether there is an integer solution to the equation $ax^2+by = c$. In NP-SPEC the program is the following (we declare that we are considering assignments to x and y in the range 10..100):

```
DATABASE
      a = 5; b = 3; c = 1874;
SPECIFICATION
      IntFunc({x,y},assign,10..100).
      fail <-- assign(x,Xval), assign(y,Yval), c != a*Xval^2 + b*Yval.
```

Finally, we present the specification for the SAT problem (cf. Subsection 2.4). In this case, we just want to guess a *subset* of the variables, and assign them the value *true*; other variables are assigned the value *false*.

```
DATABASE
      N = 100;      // number of propositional variables
      IN_CLAUSE = { // IN_CLAUSE(X,Y) <--> literal X is in clause Y
                    (12,1),(25,1),(71,2),(-23,2), ... };
SPECIFICATION
      Subset({1..N},true).
      good_clause(Y) <-- in_clause(X,Y), X > 0, true(X).
      good_clause(Y) <-- in_clause(X,Y), X < 0, NOT true(-X).
      fail <-- NOT good_clause(Y).
```

We note that predicate `good_clause` is defined, i.e., is in the head of auxiliary rules. A guessed assignment `fails` if it produces a clause which is not good; a clause is good if it contains at least a literal which is assigned a truth value that makes it true.

We remark that the declarative style of programming in NP-SPEC 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 by adding the following rule

```
      fail <-- path(X,NODES), 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`.

2.2 Formal Properties of NP-SPEC

The formal properties of NP-SPEC are explained in detail in [3]. Concerning syntax, we remark that NP-SPEC offers also useful aggregates, such as `SUM`, `COUNT`, `MIN`, and `MAX`.

The semantics of NP-SPEC is based on a generalization of the *minimal model* semantics of [26], called (P,Q)-*minimal* model semantics [20]. The formal definition can be found in [4], here we just recall some elementary notions. The *Herbrand universe* U of an NP-SPEC program S is the set of all constant symbols occurring in S. The *Herbrand base* of S is the set $\{p(e_1,\ldots,e_n) \mid p \text{ is a predicate symbol of arity } n \text{ of } S \text{ and } e_1,\ldots,e_n \in U\}$. A model of S is a subset of its Herbrand base which satisfies its rules and assigns *false* to `fail`.

As for its computational properties, the *data complexity* of NP-SPEC, i.e., the complexity of query answering measured in the size of the input extensional database only, is NP-complete. The *expressiveness* of NP-SPEC is such that the language captures NP. This has been proven showing that, for each problem A in NP, there is a *fixed* database-free NP-SPEC program SP such that for each instance *db* of A encoded as an input database DB, it holds that $SP \cup DB$ returns a solution iff *db* is a "yes" instance of A. This means that NP-SPEC is capable of specifying exactly all problems belonging to NP. We remind that, conversely, DATALOG is capable of expressing only a strict subset of the polynomial-time problems. As an example, it cannot express the "even" query, which input is a domain C of objects, and which question is: "Is the cardinality of C even?".

2.3 Prolog-Based Compilation

The first implementation of NP-SPEC has been in ECL^iPS^e [1], a PROLOG engine integrated with several extensions.

The compiler takes two files, one containing the specification section, and another containing the database section of a NP-SPEC program, and merges them with a program-independent header to form an ECL^iPS^e target program file.

The ECL^iPS^e runtime system evaluates the target program file and produces the results. The prototype implements a simple guess-and-check evaluation strategy. This is obtained by defining the search space using ECL^iPS^e constraint declaration mechanisms, and then instantiating all domain variables before proceeding with constraint checking.

This approach allowed us to obtain a fast implementation, because it relies on the mechanisms of unification typical of PROLOG. As for the efficiency, we were able to solve just toy-size instances of NP-complete problems.

2.4 SAT Technology

A propositional formula in *conjunctive normal form* (CNF) is a set of *clauses*, and a clause is a set of *literals*. A literal is either a propositional variable or the negation of a propositional variable. Sometimes a formula in CNF is referred to as a *conjunction* of clauses, and a clause as a *disjunction* of literals. The *vocabulary* $V(T)$ of T is the set of propositional variables occurring in T. An interpretation of T is an assignment of a Boolean value, i.e., either *true* or *false*, to each variable in $V(T)$. A *model* of T is an interpretation that assigns *true* to T, using the usual semantical rules for the interpretation of negation, disjunction, and conjunction. The SAT problem has as input a formula T in CNF and the question is whether T is *satisfiable*, i.e., if it has a model, or not.

Algorithms for the SAT problem are either *complete*, i.e., if there is a model, they are guaranteed to find one, or *incomplete*, i.e., they may fail to find a model if there is one.

The main complete algorithms for SAT are based on the famous DPLL procedure [9,8], and may differ quite a lot on the heuristics for the variable selection.

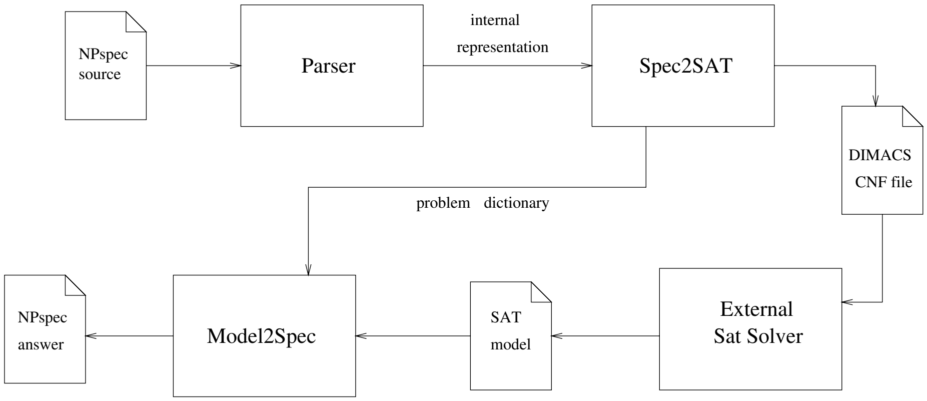


Fig. 1. Architecture of the NP-SPEC compilation and execution environment

We have performed preliminary tests with various solvers. The relatively simple mechanism of constraint propagation of DPLL can be implemented in a quite efficient way, and gives surprisingly good results. Complete algorithms are able to solve SAT instances of several hundreds of variables and several thousands of clauses in the worst conditions, i.e., at the so-called *crossover point* [25]. Such point refers to a particular random generation of CNFs, and is determined experimentally as the point in which the probability of a formula to be satisfiable equals the probability of being unsatisfiable. As for instances of SAT which are not randomly generated, the size of formulae that can be dealt with is quite larger. Generally, incomplete algorithms are much faster than complete ones. Most popular algorithms such as GSAT and WALKSAT [24] are based on randomized local search.

Many solvers are publicly available on the WWW, cf. e.g., [23], and use the DIMACS [15] input format, i.e., a text file containing one clause per line, where each line contains an integer for each literal, and is terminated by 0.

In the experiments presented in Section 4 we used the DPLL-based complete system SATZ, described in [19].

3 Compilation into SAT

Our system is written in C++, and its general architecture is shown in Figure 1.

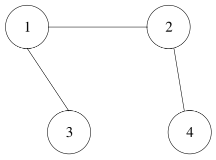
The module PARSER receives a text file containing the specification S in NP-SPEC, parses it, and builds its internal representation. The module SPEC2SAT compiles S into a CNF formula T in DIMACS format, and builds an object representing a dictionary which makes a 1-1 correspondence between ground atoms of the Herbrand base of S and variables of the vocabulary $V(T)$. The file in DIMACS format is given as an input to the SAT solver, which delivers either a text file containing 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) in the original language of the specification.

In the current version we do not allow aggregates, recursion and negative occurrences of defined predicates in NP-SPEC. It is important to note that such syntactic restrictions do not limit the expressive power of NP-SPEC [4].

3.1 Basic Algorithm of SPEC2SAT

From this point on we focus on the SPEC2SAT module, the most important of the system. Formally, the module receives as input an NP-SPEC specification $S = \langle DB, SP \rangle$, and outputs a propositional formula T in CNF such that T is satisfiable if and only if the answer to S is "yes"; moreover, if T is satisfiable, then each model of T corresponds to a solution of S.

As an example, Figure 2 shows an instance of graph 3-coloring (a), the corresponding dictionary (b) and the DIMACS file generated (c).



(a) Problem Instance

NP-SPEC atom	variable	kind
edge(1,2)	1	
edge(1,3)	2	α
edge(2,4)	3	
coloring(1,0)	4	
coloring(1,1)	5	
coloring(1,2)	6	
coloring(2,0)	7	
coloring(2,1)	8	
coloring(2,2)	9	β
coloring(3,0)	10	
coloring(3,1)	11	
coloring(3,2)	12	
coloring(4,0)	13	
coloring(4,1)	14	
coloring(4,2)	15	
fail	16	γ

(b) Dictionary

```
+-----+------------+---------------+-------+
|     | -4 -5 0    |               |       |
|     | -4 -6 0    |               |       |
|     | -5 -6 0    | 16 -1 -4 -7 0 |       |
|     | 4 5 6 0    | 16 -2 -4 -10 0 |      |
|     | -7 -8 0    | 16 -3 -7 -13 0 |      |
| 1 0 | -7 -9 0    | 16 -1 -5 -8 0 | -16 0 |
| 2 0 | -8 -9 0    | 16 -2 -5 -11 0 |      |
| 3 0 | 7 8 9 0    | 16 -3 -8 -14 0 |      |
|     | -10 -11 0  | 16 -1 -6 -9 0 |       |
|     | -10 -12 0  | 16 -2 -6 -12 0 |      |
|     | -11 -12 0  | 16 -3 -9 -15 0 |      |
|     | 10 11 12 0 |               |       |
|     | -13 -14 0  |               |       |
|     | -13 -15 0  |               |       |
|     | -14 -15 0  |               |       |
|     | 13 14 15 0 |               |       |
+-----+------------+---------------+-------+

   1        2             3            4
```

(c) CNF formula: clauses of kinds 1..4

Fig. 2. Example of the results of the SPEC2SAT module

In order to understand how the given dictionary and file are obtained, we first notice that only a subset of the Herbrand base of S is really meaningful for the compilation process. For example, ground instantiations of a predicate in DB which are not facts in the database can be neglected when building the vocabulary $V(T)$. More precisely, $V(T)$ is made of variables of three kinds:

α. one variable for every fact in DB;
β. one variable for every ground instantiation of a guessed predicate on elements of the relevant domain;
γ. one variable for every ground instantiation of other predicates.

Figure 2(b) shows the three kinds of variables for the graph coloring problem. In particular, for the *coloring* predicate we have $\mathtt{N} \cdot \mathtt{K}$ atoms of the kind β.

The set of clauses of T is made of clauses of four kinds:

1. A clause $\{c\}$ for each variable c of the kind α.
2. Clauses using variables of the kind β encoding the meaning of the corresponding metarule.
3. Clauses using variables of the kind α, β, and γ encoding the meaning of the rules of SP. Each ground instantiation of a rule can in principle originate several clauses.
4. The clause $\{\neg \mathit{fail}\}$.

Figure 2(c) shows the four sets of clauses for the current example. In particular, metarule `GC1` originates the clauses of kind 2 and rule `GC2` originates the clauses of kind 3. Clauses of kind 2 are of the following two subkinds:

$$\neg \mathit{coloring}(r, c_1) \vee \neg \mathit{coloring}(r, c_2) \quad \forall r \in \{1..n\} \;\; \forall c_1, c_2 \in \{0..k-1\} \tag{1}$$
$$\mathit{coloring}(r, 0) \vee \cdots \vee \mathit{coloring}(r, k-1) \quad \forall r \in \{1..n\} \tag{2}$$

Set of clauses (1) states that each node has at most one color in $0..k-1$ and set (2) that each node has at least one color. Similar sets of clauses exist for the other kinds of metapredicates, i.e., for `Permutation`, `IntFunc`, and `Subset`.

3.2 Optimization of Spec2SAT

Several simplifications of T are possible, some simple and some more complex. The simple ones consist in eliminating the unary clauses, i.e., those of kind 1 and 4. This implies that clauses in which such literals occur will be –according to the sign– either shortened or eliminated.

More complex optimizations –which do not apply for very simple specifications such as the one for graph coloring– involve clauses of kind 3, and are based on the elimination of useless variables of the kind γ. Let us consider for example the following rule:

```
p(X,Y,Z) <-- q(X,Y), s(Y,Z,W), r(Z,W).
```
(3)

In principle, a clause of T is generated for each of the $|U|^4$ ground instantiations of the rule (one for each possible value assignment to the four variables occurring in it). This can be obviously unpractical when the Herbrand universe U is sufficiently large, e.g., $|U| > 100$.

Our goal is to avoid most of those instantiations, by using information on the plausible extensions of the predicates. In order to do that, first of all we build the *precedence graph* G of SP. The nodes of G are the predicates of SP, and there is an edge from q to p iff there is a rule with p in the head and q in the body in SP. Predicates of SP are naturally partitioned in two subsets:

– "primitive", i.e., sources in G; they are either predicates of DB, or guessed predicates;
– "defined", i.e., they occur in the head of some rule.

Note that the special predicate `fail` is defined, and it is actually the only sink of G. Note also that G is a DAG, since recursion is not allowed.

Basically, predicates are processed in the order given by the topological sort of G, i.e., no node is visited before all of its predecessors are visited, and each predicate contributes to T with a set of clauses. In particular, primitive predicates are quite straightforward to deal with, and they are taken into account by the clauses of kinds 1 and 2.

As for a defined predicate p, using the assumption that it is considered only after all predicates occurring in the body of rules with p in the head have been considered, in some important cases we can discard several instantiations of such rules. As an example, in rule (3) if `q` is a DB predicate we have to consider only value assignments to `X` and `Y` which correspond to facts in DB, instead of all $|U|^2$ assignments.

Generalizing this idea, we introduce the notion of "alive" for ground instantiations of predicates. In particular, the set $alive(p)$ (a subset of the Herbrand base) of ground instances of p is defined in the following way:

– if p is a primitive predicate, $alive(p)$ is the set of the ground instantiations corresponding to variables of the kinds α and β;
– if p is a defined predicate, $alive(p)$ is recursively defined as the set of atoms occurring in the head of ground instances of rules with p in the head, such that all positive literals in the body are alive.

Our algorithm traverses G following its topological sort. When a predicate p is under analysis, the set $alive(p)$ is built, and clauses (kind 3) corresponding to instantiations of rules with p in the head are generated. Referring to the above example, in rule (3) we must consider only value assignments to the four variables according to $alive(q)$, $alive(s)$, and $alive(r)$.

Another key point of the algorithm concerns the way assignments to the variables are generated. As we already mentioned the number n of variables in a rule is a crucial parameter, because in the worst case $|U|^n$ variable assignments must be taken into account. In very simple specifications, n can be as large as 10 (cf. Section 4.3), and $|U|$ as large as 100, therefore it is very important to avoid a simple-minded enumeration of all variable assignments.

To this aim we use a backtracking-based algorithm that for each rule explores a tree of depth n (one level for each variable) and uses sets *alive* for pruning the search. The algorithm considers *partial* assignments, i.e., assignments to a subset of m variables, with $m < n$. As an example, referring to rule (3), if `Z` and `W` have already been assigned to, e.g., 2 and 7, and $\mathtt{r}(2, 7) \not\in alive(\mathtt{r})$, then it is useless to consider assignments to the other variables `X` and `Y`.

4 Considerations on Performance

In this section we discuss the effectiveness of the system for the solution of NP-complete problems. It is quite obvious that, in terms of performances, our system cannot compete with state-of-the-art solvers of the original problems. In fact, our system is meant mainly for developing executable specifications, rather than for effective program development. The main emphasis of our work is on obtaining simple and readable specifications, an activity that in NP-SPEC typically takes hours or even minutes; on the other hand implementing a very efficient solver for a new problem in NP may take weeks or even months. Nevertheless, we want to show that the system is able to solve medium-size instances of various classical hard problems.

Conversely, the enumerative algorithm of the original PROLOG engine of NP-SPEC is able to solve only small instances. We believe that the ability to solve non-toy cases helps the user to get a better understanding of her/his application and to capture aspects of the problem that might fail to appear in very small instances.

In the following subsections, we analyze the performances of our system on three problems: graph coloring, Hamiltonian cycle, and job shop scheduling. Experiments use the solver SATZ and run on a Pentium II PC at 300 MHz. Times are expressed in seconds of CPU use, and the symbol "–" means that SATZ did not terminate within half an hour.

The total time for finding a solution is the sum of the compilation time t_1 and the time t_2 needed by SATZ. We remark that, asymptotically, t_1 is polynomial in the size of the input, while t_2 can be exponential in the worst case. Nevertheless, in some cases $t_1 > t_2$, as an example when the generated CNF is quite large and has many models.

4.1 Graph Coloring

The specification of the graph coloring problem has been provided in Section 2.1.

Given a graph G with n nodes, e edges, and k colors, the compilation of the metarule `GC1` generates a formula with $n \cdot k$ variables, one for each pair $(\langle node \rangle, \langle color \rangle)$. The formula contains $O(n \cdot k^2)$ clauses which state that each node has exactly one color. The rule `GC2` adds $e \cdot k$ clauses that forbid two adjacent nodes to have the same color (cf. Figure 2(c)).

For the experimentation of the system, we use a set of instances taken from the DIMACS benchmark repository. In particular, we select the family DSJC of randomly-generated graphs proposed in [14]. Table 1 reports our results.

Table 1. Performances on the graph coloring problem

graph	nodes	edges	colors (min)	colorable	compile time	SAT time	variables	clauses
125.1	125	736	4	NO	3.19	0.33	500	3,819
125.1	125	736	5 (5)	YES	3.27	0.20	625	5,055
125.5	125	7,782	21 (16)	YES	26.62	54.10	2,625	108,086
250.1	250	3,218	9 (9)	YES	30.53	4.12	2,250	38,212
250.5	250	31,336	40 (27)	YES	234.1	215.6	10,000	821,970
500.1	500	12,456	16 (14)	YES	243.52	63.91	8,000	259,828

The table shows that the system has been able to solve some large instances. In one case, i.e., the instance DSJC.125.1, it has been able also to prove the minimality of k. Such a result has been obtained by proving the unsatisfiability of the formula generated with $k-1$ colors. Conversely, for some instances it found a solution only for a less constrained instance with a number of colors larger than the minimum (provided in parenthesis).

4.2 Hamiltonian Cycle

The specification of the Hamiltonian cycle problem has also been provided in Section 2.1.

Given a graph G with n nodes and e edges, the resulting SAT formula has n^2 variables and $O(n^3)$ clauses. In fact, the compilation generates a variable for each fact of the form `path(i,j)`, with `i` and `j` ranging from 1 to n. The number of clauses generated by the metapredicate `Permutation` is $O(n^3)$. The number of clauses generated by the rules `H2` and `H3` depends on e. In the two extreme cases of complete and empty graph, no clauses and exactly n^3 clauses, respectively, are generated.

We experiment our system on random instances. It is known [6] that the hardest random instances are obtained for a number of edges e equal to $p = n \log n/2$, i.e., p is the crossover point. We consider graphs such that $e = p$, taking into account both solvable and unsolvable instances. In addition, we consider graphs far from that point: solvable instances with $e = 3p/2$, and unsolvable ones with $e = p/2$.

Table 2 shows our average results of 5 instances for $n = 15$, 17, and 20 ($p = 60$, 70, and 86, respectively). We note that compilation is quite fast, while the SAT solver is very slow. In fact, the solver is able to handle easily only satisfiable instances with $n = 15$ and 17. Unsatisfiable instances are solved very slowly by SATZ, even for "easy" instances for the original problem. For larger instances, the solver is quite inefficient for the satisfiable cases, and ineffective for unsatisfiable ones.

Experiments with other SAT solvers provide similar results, whereas current solvers of the Hamiltonian cycle problem are able to solve instances with thousands of nodes quite easily.

Table 2. Performances on the Hamiltonian cycle problem

nodes	edges	cycle	compile time	avg. SAT time	variables	clauses
15	30	NO	1.00	478.47	225	6,570
15	60	NO	1.00	1,577.00	225	6,090
15	60	YES	1.00	0.53	225	6,090
15	90	YES	1.00	0.39	225	5,190
17	35	NO	1.33	1,342.87	289	8,976
17	70	NO	1.33	–	289	8,364
17	70	YES	1.33	4.03	289	8,364
17	105	YES	1.33	1.09	289	7,752
20	43	NO	2.73	–	400	14,780
20	86	NO	2.73	–	400	13,900
20	86	YES	2.73	704.61	400	13,900
20	130	YES	2.73	324.41	400	13,020

4.3 Job Shop Scheduling

Job shop scheduling [12, Prob. SS18, p. 242] is a very popular NP-complete scheduling problem. In job shop scheduling, there are n jobs, m tasks, and p processors. Jobs are ordered collections of tasks and each task has a length and the processor that performs it. Each processor can perform a task at the time, and the tasks belonging to the same job must be performed in their order. Finally, there is a global deadline D that has to be met by all jobs. In NP-SPEC the problem is specified as follows.

```
DATABASE
   TASKS = 36;  D = 55;
      // task(T,J,Po,Pr,L): the task T belongs to job J in position Po,
      // it runs on processor Pr with length L
   task = {(1,1,1,2,1), (2,1,2,6,3), (3,1,3,1,6), ..., (36,6,6,2,1)};
SPECIFICATION
      // start_time(T,S): task T starts at time S
   IntFunc({1..TASKS},start_time,0..D-1).
       // tasks T1 and T2 of job J are ordered correctly
   fail <-- start_time(T1, S1),  task(T1, J, Po, _, L1),
            start_time(T2, S2),  task(T2, J, Po + 1, _, _),
            S2 < S1 + L1.
      // no overlap of tasks in the same processor
   fail <-- start_time(T1, S1),  task(T1, _, _, Pr, L1),
            start_time(T2, S2),  task(T2, _, _, Pr, L2),
            T1 != T2,  S1 <= S2,  S2 < S1 + L1.
   fail <-- start_time(T1, S1),  task(T1, _, _, Pr, L1),
            start_time(T2, S2),  task(T2, _, _, Pr, L2),
            T1 != T2,  S2 <= S1,  S1 < S2 + L2.
      // meet the deadline
   fail <-- start_time(T1, S1),  task(T1, _, _, _, L1),
            L1 + S1 > D.
```

The compilation of this NP-SPEC file generates a SAT instance with $m \cdot D$ variables. Regarding the number of clauses, for each quadruple $\langle t_1, t_2, i, j\rangle$ formed by two tasks t_1 and t_2 and two time points i and j, there is a clause if and only if one of the rules prohibits t_1 to start at i and t_2 to start at j jointly. This number of clauses is $O(m^2 \cdot D^2)$, and its actual value depends on the relative length of the tasks which belong to either the same job or the same processor.

Many benchmark instances are available for this problem, whose sizes range from 36 to 1,000 tasks. We consider two relatively small instances, known as FT06 (36 tasks, 6 jobs, 6 processors, solvable with deadline 55), and LA02 (50 tasks, 10 jobs, 5 processors, solvable with deadline 655).

As shown in Table 3 the first instance is solved easily, and the proof of the optimality of the deadline (i.e., no solution with deadline 54) is quite fast as well. Unfortunately, for the second instance, the SAT instance generated has more than a billion clauses, and it is too big to be solved by the current solvers. In order to find at least an approximate solution, we create a new instance called LA02r in which all lengths are divided by 20 and rounded up. This corresponds to reducing the granularity of the problem, and allowing only starting times divisible by 20. The smallest deadline found for LA02r is $D = 46$, which corresponds to $920 (= 46 \cdot 20)$ in LA02. If we give a looser deadline of 1,000, the problem is solved much faster. We remark that the minimum value of the deadline for this instance is 655, and all state-of-the art solvers find solutions below 700.

Table 3. Performances on the job shop scheduling problem

instance	tasks	deadline	solvable	compile time	SAT time	variables	clauses
FT06	36	54	NO	215.07	53.07	1,944	355,871
FT06	36	55	YES	220.17	20.140	1,980	365,333
LA02	50	920	YES	335.830	364.74	2,300	392,816
LA02	50	1,000	YES	426.71	19.390	2,500	440,628

Summing up, these results show that for job shop scheduling, the critical factors are the compilation times and the size of the SAT formula obtained. Conversely, the solution of such formula is relatively fast compared with its size.

We remark that, without the optimizations described in Section 3.2, none of the instances of Table 3 was compiled by SPEC2SAT in less than one day.

5 Conclusions, Related, and Future Work

We have presented a novel approach for the execution of specifications of problems in NP, based on the translation into SAT. The performance of the resulting system is very good, compared to the previous, PROLOG-based, engine underlying NP-SPEC. As an example, we were able to solve benchmarks of graph coloring problems with 500 nodes, while the previous approach was able to deal with graph of just 14 nodes. As another example, we were able to increase the size of the chessboard in the n-queens problem –in which the goal is to place n non-attacking queens on a $n \times n$ chessboard– from 12 to 60. The reason for such an increase in performance is that we exploit the best SAT solvers, developed by third parties. Further improvements of SAT solvers will reflect in an improvement of our system.

Moreover, our system can be used as a tool for the generation of new benchmark instances of SAT. In fact, SAT solvers are currently tested on encodings of a variety of problems, such as graph coloring, planning, Latin square, blocks world, Towers of Hanoi, circuit fault analysis, and others [23]. For example, the encoding used for graph coloring is the same as the one generated by SPEC2SAT with our specification of Section 2.

As for related research, we have listed in the introduction several approaches to the solution of problems, ranging from planning to cryptography, based on translation into SAT.

Other researchers [10,22] propose DATALOG-like languages for problem specification. The main difference between NP-SPEC and the other languages relies in its semantics, which is based on the notion of model minimality. Alloy Analyzer, a system for reasoning in an extension of first-order logic based on a translation to SAT, has been proposed in [13]. The main difference wrt NP-SPEC is that in Alloy Analyzer in general decidability is not guaranteed, and consequently the user must supply a bound on the number of atoms in the universe.

In the future, first we plan to include aspects of NP-SPEC that have been neglected in this version, such as aggregates and recursion. Furthermore, we plan to introduce some form of program transformation for improving compilation in terms of both the size and the hardness of the generated formula. Finally, we want to equip the system with a learning mechanism for automatic selection of the best SAT solver for the instance at hand. In particular, fast incomplete algorithms could be used for instances that are known to be satisfiable.

Acknowledgements. The authors are grateful to Giovambattista Ianni, Luigi Palopoli, and Domenico Vasile for useful discussion and for providing some examples.

References

1. A. Aggoun *et al.* *ECL^{i}PS^e User Manual (Version 4.0).* IC-Parc, London (UK), July 1998.
2. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, Los Altos, 1988.
3. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. Technical Report 00-13, Dip. di Inf. e Sist., Univ. di Roma “La Sapienza”, 2000.
4. M. Cadoli and L. Palopoli. Circumscribing DATALOG: expressive power and complexity. *Theor. Comp. Sci.*, 193:215–244, 1998.

5. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In *Proc. of PADL'99*, number 1551 in LNAI, pages 16–30. Springer-Verlag, 1999.
6. P. Cheeseman, B. Kanefski, and W. M. Taylor. Where the really hard problem are. In *Proc. of IJCAI'91*, pages 163–169, 1991.
7. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of AAAI'94*, pages 1092–1097, 1994.
8. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5(7):394–397, 1962.
9. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of the ACM*, 7:201–215, 1960.
10. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proc. of KR'98*, pages 406–417, 1998.
11. M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. of IJCAI'93*, pages 52–57, 1993.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness.* W.H. Freeman and Company, San Francisco, 1979.
13. D. Jackson. Automating first-order relational logic. In *Proc. of ACM SIGSOFT'00: 8th SIGSOFT Symposium on Foundation of Software Engineering*, 2000.
14. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.
15. D. S. Johnson and M. A. Trick, eds. *Cliques, Coloring, and Satisfiability. Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Math. Soc., 1996.
16. H. A. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. of KR'96*, pages 374–384, 1996.
17. H. A. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI'92*, pages 359–363, 1992.
18. T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on Computer-Aided Design*, pages 4–15, 1992.
19. C. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of IJCAI'97*, pages 366–371, 1997.
20. V. Lifschitz. Computing circumscription. In *Proc. of IJCAI'85*, pages 121–127, 1985.
21. F. Massacci and L. Marraro. Logical cryptanalysis as a SAT-problem: Encoding and analysis of the U.S. Data Encryption Standard. *J. of Automated Reasoning*, 24(1-2):165–203, 2000.
22. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. of Mathematics and Artif. Intell.*, 25(3,4):241–273, 1999.
23. SATLIB - The Satisfiability Library.
`http://www.informatik.tu-darmstadt.de/AI/SATLIB`.
24. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of AAAI'94*, pages 337–343, 1994.
25. B. Selman, D. Mitchell, and H. Levesque. Generating Hard Satisfiability Problems. *Artificial Intelligence*, 81:17–29, 1996.
26. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. of the ACM*, 23(4):733–742, 1976.