Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 162 (2005) 89–120

Artificial Intelligence

www.elsevier.com/locate/artint

Compiling problem specifications into SAT ☆

Marco Cadoli [a,*], Andrea Schaerf [b]

[a] *Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy*

[b] *Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica, Università di Udine, Via delle Scienze 208, I-33100 Udine, Italy*

Received 20 November 2001; accepted 21 January 2004

Available online 10 December 2004

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 as to capture 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 performance of the system has been tested on a few classical problems, namely graph coloring, Hamiltonian cycle, job-shop scheduling, and on a real-world scheduling application, namely the tournament scheduling problem.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Automatic generation of problem reformulation; Executable specifications; SAT problem; NP-complete problems

☆ This paper is an extended and revised version of "Compiling problem specifications into SAT", which appeared in Proceedings of the European Symposium on Programming (ESOP 2001), Lecture Notes in Computer Science, vol. 2028, Springer, Berlin, 2001, pp. 387–401.

* Corresponding author.

E-mail addresses: cadoli@dis.uniroma1.it (M. Cadoli), schaerf@uniud.it (A. Schaerf).

URLs: http://www.dis.uniroma1.it/~cadoli (M. Cadoli), http://www.diegm.uniud.it/schaerf (A. Schaerf).

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 syntax, i.e., PROLOG with no function symbols. 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 [1]. NP-SPEC allows the user to express every problem belonging to the complexity class NP [2], which includes many notorious 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, the system constructs the solution of π from the variable assignments that satisfy T.

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 to avoid the generation of useless clauses insofar as possible.

The approach of translation into SAT is motivated by the huge amount of research devoted to such a problem in recent years (see, e.g., [3]), and the number of fast solvers available from the research community. Such solvers, both complete and incomplete ones, are able to solve instances of hundreds of thousands of clauses in a few seconds, a result inconceivable only a few years ago. In addition, the community working on SAT is still very active, and better and better SAT solvers are expected 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 for a number of years in problems such as planning [4–6], scheduling [7], theorem proving in finite algebra [8], generation of test patterns for combinatorial circuits [9], and cryptography [10]. Those papers showed that translating a problem into SAT can give good performance in the resulting system, as compared with state-of-the-art dedicated solvers.

The shortcoming of those previous works is that the translator 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 performance, NP-SPEC obviously cannot outperform state-of-the-art solvers of well-studied problems. However, 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 medium-size instances of various classical problems in reasonable time. In addition, it works much faster than the original NP-SPEC engine [11] which is based on a translation of the input specification in the logic programming language PROLOG.

The paper is organized as follows. In Section 2 we introduce the language NP-SPEC and recall the state of the art on SAT technology. In Section 3 we describe the compiler. In Section 4 we illustrate the performance of the system in four problems: graph coloring, Hamiltonian cycle, job-shop scheduling, and tournament scheduling. Related work is discussed in Section 5. Finally, in Section 6 we draw conclusions and discuss future work.

2. Preliminaries

2.1. *Overview of the* NP-SPEC *language*

As a first example, we show an NP-SPEC program for the *Hamiltonian path* NP-complete problem [2, 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 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.

More formally, an NP-SPEC program consists of a `DATABASE` section and a `SPECIFICATION` section (cf. Appendix A for the complete syntax). The `DATABASE` section includes:

- Definition of extensional relations of the kind

  `r = {t1,...,tn}`,

  where `r` is the input relation name and each `ti` is a tuple of the same arity as `r`. The only constant symbols allowed in tuples are integers and strings.
- Definition of constants.

The `SPECIFICATION` section consists of two parts:

- A *search space* declaration, which corresponds to the definition of the domain of the guessed predicates. In basic NP-SPEC it is a sequence of declarations of the form:
  (1) `Subset(<domain>, <predicate_id>).`
  (2) `Permutation(<domain>, <predicate_id>).`
  (3) `Partition(<domain>, <predicate_id>, n).`
  (4) `IntFunc(<domain>, <predicate_id>, min..max).`
  where `<predicate_id>` is the name of the guessed predicate and `<domain>` is a finite set defined either as an input relation, or as an enumeration, or by means of union (‘`+`’), intersection (‘`*`’), difference (‘`-`’), and Cartesian product (‘`><`’).

  `<domain>` identifies the domain upon which the extension of the predicate is guessed, and, for `Subset`, it must have the same arity as `<predicate_id>`. In the other cases `<predicate_id>` is a guessed predicate of arity equal to the arity a of `<domain>` plus 1. Such a declaration means that `<predicate_id>` can have all extensions such that the first a arguments coincide with a member of `<domain>`, while the last one depends on the metapredicate. In particular:
  – For `Permutation` the extension of `<predicate_id>` must represent a bijective function from `<domain>` to the interval $\{1..c\}$, where c is the cardinality of `<domain>`.
  – Declarations using metapredicate `Partition` have a further integer-valued argument `n` that states the number of subsets in which the domain must be partitioned. The extension of `<predicate_id>` must represent a function from `<domain>` to the interval `{1..n}`, the last argument being any element of such an interval.
    `Subset` is indeed the special case of `Partition` in which `n = 2`, but the syntax is different (declaration `Subset(<domain>, <predicate_id>)` implies that the arity of `<predicate_id>` equals the arity of `<domain>`).
  – The metapredicate `IntFunc` is a generalization of `Partition` and can be used to model functions from `<domain>` to the interval `{min..max}`, `min` and `max` being two integers.
    Since `IntFunc` generalizes both `Partition` and `Subset`, the latter are not strictly necessary. Nevertheless, their usage improves readability of a specification.
– A *stratified* DATALOG program that can possibly include the six predefined relational operators and negative literals, which have the form `NOT A`, where `A` is an atom. Strat-

ification introduces a restricted form of negation by allowing negative literals in the body of rules [12]. Intuitively, a program is stratified if a predicate is not defined recursively through negation. More formally, its *precedence graph* must not contain a cycle with a negative edge. Such a graph contains a node for each predicate symbol, and a positive (negative) edge from node A to node B if A occurs positively (negatively) in the body of a rule with B in the head.
This part includes also the rules necessary for specifying a problem, i.e., one or more rules with `fail` in the head.

Another example concerns the famous n-queens problem, in which the goal is to place n non-attacking queens on an $n \times n$ chessboard. The NP-SPEC program for the 12-queens is the following:

```
DATABASE
  nb_queens = 12;
SPECIFICATION
  Permutation({1..nb_queens}, queens).    // queens(R,C) <-> there
                                       // is a queen in row R,column C
  fail <-- queens(R1,C1), queens(R2,C2),  // no 2 queens attacking
           R1 != R2, R1 - R2 == C1 - C2.  // on SE-NW diagonals
  fail <-- queens(R1,C1), queens(R2,C2),  // no 2 queens attacking
           R1 != R2, R1 - R2 == C2 - C1.  // on NE-SW diagonals
```

Also in this case, we declare `queens` to be a guessed predicate of arity 2 in the search space declaration. Declaring it as a permutation rules out attacks on the same row or on the same column. The example shows that arithmetical operators are allowed.

As another example, in the *graph coloring* NP-complete problem [2, Prob. GT4, p. 191] the input is 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;  // no. of colors
      n = 6;  // no. of nodes
      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 problem [2, Prob. AN8, p. 250] the input is given by 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 3-SAT problem, which is the SAT problem [2, Prob. L01, p. 259] (cf. Section 2.4), in which each clause has exactly three literals. In this case, we just want to guess a *subset* of the variables, and assign them the value *true*; other variables are implicitly assigned the value *false*.

```
DATABASE
  n = 100;
    // clause(L1,L2,L3): L1, L2, and L3 are the literals of
    // a clause
  clause = {(21, -40, -11), (37, 88, -6), (28, -1, 94), ...};

SPECIFICATION
  Subset({1..n},true).
    // val(L,V,S): L is a literal, V is the corresponding
    //             variable, and S the opposite of the sign of L,
    //             i.e., if V has truth value S, then L is false
    //
  val(L,L,0) <-- L > 0.
  val(L,V,1) <-- L < 0, V == -L.
  assign(V,0) <-- NOT true(V).
  assign(V,1) <-- true(V).
  fail <-- clause(L1,L2,L3),
           val(L1,V1,S1), val(L2,V2,S2), val(L3,V3,S3),
           assign(V1,S1), assign(V2,S2), assign(V3,S3).
```

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 [2, 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 NP-SPEC offers also useful SQL-style aggregates, such as `SUM`, `COUNT`, `MIN`, and `MAX`. They are not discussed in this paper, because they are not considered in the current implementation, see [11] for their syntax and semantics.

2.2. Formal properties of NP-SPEC

The formal properties of NP-SPEC are explained in detail in [11] and are briefly recalled here. The semantics of an NP-SPEC program can be explained as a two-step process. First of all, all syntactic sugar is eliminated from the specification, and the result is a set of metarules containing only the `Subset` metapredicate and a corresponding reformulation of the rules. As an example concerning the specification of the n-queens problem shown before, the metarule containing the `Permutation` metapredicate must be rewritten using only `Subset`, and this can be done by guessing all possible pairs consisting of a row and a column. Since now it is not guaranteed that there are no two queens attacking on the same row or column, a set of appropriate rules enforcing such constraints must be added.

As for the second step, we must briefly recall a couple of 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 $\{p(e_1, \ldots, e_n) \mid p$ is a predicate symbol of arity n of S and $e_1, \ldots, e_n \in U\}$. An *interpretation* of S is a subset of the Herbrand base, and an interpretation that satisfies all rules of S, with the obvious meaning of implication, is a *model* of S. Some models are selected if they are *minimal* with respect to a specific criterion, which is based on a generalization of the well-known *minimal model* semantics defined in [1], called (P, Q)*-minimal* model semantics [13]. Intuitively, the (P, Q)*-minimal* model semantics introduces a partial order between interpretations, in which only interpretations assigning the same extension to guessed predicates are comparable: this guarantees that all possible choices for a guessed predicate are taken into account. The partial order states that, between two comparable extensions, the one with smaller extension for some non-guessed predicate must be preferred: this guarantees that atoms in the head of rules become true only if there is a reason. Finally, only (P, Q)-minimal models of S which assign *false* to `fail`, if any, are considered: this guarantees that all constraints in S are satisfied. The answer of S is any of such models.

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 formally proven in [11] by resorting to Fagin's famous theorem [14], which states that the specification of every problem in the NP complexity class can be stated as a formula of existential second-order logic (ESO) interpreted on a finite database and, conversely, that every formula of ESO corresponds to the specification of a problem in NP. The proof shows that, for each ESO formula (problem in NP) A, 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 if and only if *db* is a "yes" instance of A.

A consequence of this fact is that the specification part of NP-SPEC is capable of specifying exactly all problems belonging to NP. We note that, conversely, DATALOG is capable of specifying only a strict subset of the polynomial-time problems. For 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 was in *ECL^{i}PS^e* [15], a PROLOG engine integrated with several extensions. The compiler takes two files, one containing the specification section S and another containing the database section D of an NP-SPEC program, analyzes them, and produces a description of the search space and of the constraints corresponding to S, and a list of facts corresponding to D, both in PROLOG format. Such a PROLOG file is merged with a program-independent header to form an *ECL^{i}PS^e* target program file.

The program-independent header contains PROLOG rules for traversing the search space in a purely enumerative fashion, with no constraint propagation and no pruning. At every point in the search space, the constraints corresponding to the rules of the NP-SPEC program are evaluated. As an example, for the search space of the n-queens problem, the *ECL^{i}PS^e* runtime system starts generating all $n!$ possible permutations, and stops as soon as it finds a permutation such that the constraints are satisfied.

This approach allowed us to obtain a fast implementation, because it relies on the unification mechanisms typical of PROLOG. As an example, it was easy to implement arithmetical expressions, because they can be readily translated into *ECL^{i}PS^e*. As for the efficiency, we were able to solve only toy-size instances of NP-complete problems, owing to the naivety of the algorithm for traversing the search space.

In the present paper we are presenting an approach which differs both in the search space that is considered, and in the way it is traversed. As an example—more details are given in the following sections—the n-queens problem will be automatically translated into a propositional formula containing n^2 variables, whose satisfiability is checked using a SAT solver. The search space is composed of all possible assignments to such propositional variables, and it is traversed in a much more efficient manner, using the powerful constraint propagation mechanism typical of SAT solvers.

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 formula 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 [16, 17], and may differ quite a lot on the heuristics for the variable selection. 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* [18]. Such point refers to a particular random generation of CNFs, and is determined experimentally as the point in which the probability of a formula being satisfiable equals the probability of being unsatisfiable. As for instances of SAT which are not randomly generated, the size of the formulae that can be dealt with is typically quite larger. Generally, incomplete algorithms are much faster than complete ones. Most popular algorithms such as GSAT and WALKSAT [19] are based on randomized local search.

Many solvers are publicly available on the WWW, cf., e.g., [20], and use the DIMACS [3] input format, i.e., a text file containing one clause per line, where each line contains a positive (resp. negative) integer for each positive (resp. negative) literal, and is terminated by `0`.

Although we have performed preliminary tests with various solvers, in the experiments presented in Section 4 we used the DPLL-based complete system SATZ, described in [21].

3. Compilation into SAT

Our system is entirely written in C++, and its general architecture is shown in Fig. 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 propositional 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) into 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

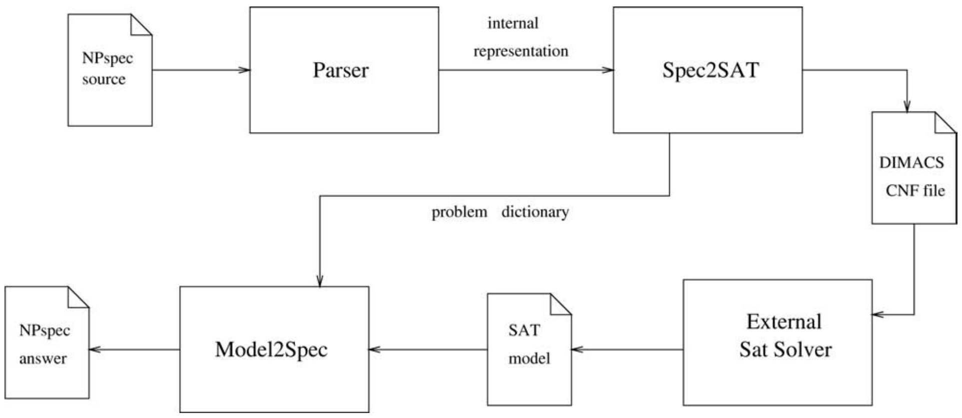


Fig. 1. Architecture of the NP-SPEC compilation and execution environment.

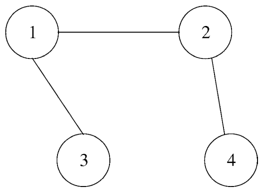
not limit the expressive power of NP-SPEC, which still express the complexity class NP [22] (although readability and efficiency of a specification may be affected.)

In the rest of this section we focus on the SPEC2SAT module, the most important of the system. We start from a general description of the algorithm, then present the main optimizations that have been introduced.

3.1. Basic algorithm of SPEC2SAT

Formally, the SPEC2SAT 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, each model of T corresponds to a solution of S.

As an example concerning graph coloring (cf. Section 2.1), Fig. 2 shows an input graph which is supposed to be 3-colored (a), the corresponding dictionary (b), and the DIMACS file generated (c). In this case we have $n = 4$ nodes and $k = 3$ possible colors (we have chosen a number of colors greater than the minimum, to make the translation more interesting). The graph is obviously 3-colorable, and the corresponding SAT instance is indeed satisfied by, e.g., assigning *true* only to propositional variables 1, 2, 3, 4, 8, 11, and 13, i.e., by coloring nodes 1 and 4 with color 0, and nodes 2 and 3 with color 1.



(a)

a	b	c	d
1 0 2 0 3 0	-4 -5 0 -4 -6 0 -5 -6 0 4 5 6 0 -7 -8 0 -7 -9 0 -8 -9 0 7 8 9 0 -10 -11 0 -10 -12 0 -11 -12 0 10 11 12 0 -13 -14 0 -13 -15 0 -14 -15 0 13 14 15 0	16 -1 -4 -7 0 16 -2 -4 -10 0 16 -3 -7 -13 0 16 -1 -5 -8 0 16 -2 -5 -11 0 16 -3 -8 -14 0 16 -1 -6 -9 0 16 -2 -6 -12 0 16 -3 -9 -15 0	-16 0

(c)

NP-SPEC atom	var.	kind
edge(1,2)	1	α
edge(1,3)	2	
edge(2,4)	3	
coloring(1,0)	4	β
coloring(1,1)	5	
coloring(1,2)	6	
coloring(2,0)	7	
coloring(2,1)	8	
coloring(2,2)	9	
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)

Fig. 2. Example of the output of the SPEC2SAT module. (a) Problem instance. (b) Dictionary. (c) CNF formula: clauses of kinds a, b, c, d.

Obtaining the vocabulary $V(T)$ starting from the specification S is, in principle, easy: we need to compute the Herbrand base of S and then define a set of propositional variables 1-1 with it. However, such a vocabulary would be unnecessarily large in many cases. For example, ground instantiations of a predicate in *DB* which are not facts in the database can be neglected when building the vocabulary $V(T)$. The basic idea for the construction of $V(T)$ (more complex techniques will be described in the next subsection) is to consider propositional 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.

Fig. 2(b) shows the three kinds of variables for the graph coloring instance. In particular, for the *coloring* predicate we have $n \cdot k = 12$ atoms of the kind β, and the only predicate which is neither in *DB* nor guessed is *fail*.

We now turn our attention to the set of clauses of T. Such clauses belong to four distinct classes:

(a) A clause $\{c\}$ for each variable c of the kind α.
Such clauses just state that every fact in *DB* is true.
(b) Clauses using variables of the kind β encoding the meaning of the corresponding metarule.
Such clauses express in propositional logic the structure of the search space, e.g., that it is a permutation.
(c) Clauses using variables of the kind α, β, and γ encoding the meaning of the rules of *SP*.
Such clauses express the constraints of the specification.
(d) The clause $\{\neg \mathit{fail}\}$, which states that we are looking for truth assignments in which all constraints are satisfied.

As for clauses of kind c, if the body of the rule contains a relational operator, such as '`==`' or '`>`' (cf. the quadratic Diophantine equations problem in Section 2.1), the expression containing the operator is evaluated and processed: if it is *true* the clause is generated, otherwise the clause is discarded.

Fig. 2(c) shows the four sets of clauses for the current example. In particular, rule `GC2` originates the clauses of kind c, and metarule `GC1` (i.e., `Partition(1..n,coloring, k)`) originates the clauses of kind b, which are of the following two subkinds:

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

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

Set (1) states that each of the n nodes has at most one color in $\{0, \ldots, k-1\}$, and set (2) that each node has at least one color in the same interval. In general, a metarule like `Partition(<domain>, <predicate_id>, k)` generates $n \cdot k$ variables and $O(n \cdot k^2)$ clauses like (1) and (2), n being the size of `<domain>`.

As for the other metapredicates, i.e., `Permutation`, `IntFunc`, and `Subset`, there are appropriate sets of variables of kind β and clauses of kind `b` encoding their meaning in propositional logic. As an example, the metarule `H1`, i.e., `Permutation({1..n}, path)`, of the Hamiltonian path specification (cf. Section 2.1) originates the following sets of clauses of kind `b`:

$$path(r,1) \vee \cdots \vee path(r,n) \quad \forall r \in \{1..n\}, \tag{3}$$

$$\neg path(r,c_1) \vee \neg path(r,c_2) \quad \forall r, c_1, c_2 \in \{1..n\},\ c_1 \neq c_2, \tag{4}$$

$$\neg path(c_1,r) \vee \neg path(c_2,r) \quad \forall r, c_1, c_2 \in \{1..n\},\ c_1 \neq c_2. \tag{5}$$

Set (3) states that each of the n nodes has at least one of the n places in the order. Set (4) states that each node has at most one place. Finally, set (5) states that each place has at most one node. Since these clauses logically imply that each place has at least one node, there is no need to explicitly state it. In general, a metarule like `Permutation(<domain>, <predicate_id>)` generates n^2 variables and $O(n^3)$ clauses like (3)–(5), n being the size of `<domain>`.

Alternative ways to encode problems based on permutations and similar structures will be discussed in Section 5.

For the sake of completeness, we show also the set of variables and clauses generated by the other metarules. A metarule like `IntFunc(<domain>, <predicate_id>, min..max)` generates the following sets of clauses:

$$\neg predicate\_id(r,c_1) \vee \neg predicate\_id(r,c_2) \quad \begin{array}{l} \forall r \in \{1..n\}, \\ \forall c_1, c_2 \in \{min..max\}, \\ c_1 \neq c_2, \end{array} \tag{6}$$

$$predicate\_id(r,min) \vee \cdots \vee predicate\_id(r,max) \quad \forall r \in \{1..n\}. \tag{7}$$

The metarule generates $n \cdot (max - min + 1)$ variables and $O(n \cdot (max - min + 1)^2)$ clauses like (6) and (7), n being the size of `<domain>`. A complex example involving such a metapredicate will be given in Section 4.3.

Finally, metarule `Subset(<domain>, <predicate_id>)` (cf., e.g., the 3-SAT example in Section 2.1) generates n variables and no clauses, since it does not imply any constraint.

3.2. *Optimization of* SPEC2SAT

In this subsection we describe some of the optimizations that we implemented in order to obtain a propositional formula T which is as small as possible in the shortest time.

First of all, there are some simplifications of T which are rather obvious: as an example, unary clauses, i.e., those of kinds `a` and `d`, can be eliminated (or, better yet, not generated at all). Clearly, this implies that clauses in which such literals occur will be—according to the sign—either shortened or eliminated.

3.2.1. Reducing the number of clauses

The first optimizations we consider involve clauses of kind `c`, and are based on the elimination of useless variables of the kind γ. Since such optimizations do not apply for very simple specifications, such as the one for graph coloring, we now turn to more abstract examples. For example, let us consider the following rule:

```
p(X,Y,Z) <-- q(X,Y), s(Y,Z,W), r(Z,W).                    (8)
```

Since there are four distinct NP-SPEC variables occurring in the rule, in principle T should contain one clause for each of the $|U|^4$ ground instantiations. This can be obviously impractical 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 plausible extensions of the predicates. As an example, we could rule out many instantiations of the rule, if we knew all *relevant* instantiations of predicate `s`. Unfortunately, it is possible to easily know the relevant instantiations only for predicates in *DB* (corresponding to variables of the kind α) and for guessed predicates (kind β).

For the defined predicates (corresponding to variables of the kind γ) we use auxiliary data structures. First of all we build the *dependency graph G* of *SP*. The nodes of G are the predicates of *SP*, and there is an edge from q to p if and only if in *SP* there is a rule with p in the head and q in the body. Since relational operators are evaluated and processed with a specific mechanism, they do not correspond to nodes. Predicates of *SP* are then 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. As an example, in Fig. 3 we show the dependency graph for the specification of the 3-SAT problem (cf. Section 2.1).

Basically, predicates are processed one at a time. Each predicate contributes to $V(T)$ with a set of propositional variables and, indirectly, to T with a set of clauses. The order in which they are processed is given according to the topological sort of G, i.e., no node is processed before all of its predecessors are processed, (cf., e.g., [23]). In particular, primitive predicates are processed first. They are quite straightforward to deal with, and they are taken into account by the clauses of kinds `a` and `b`.

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

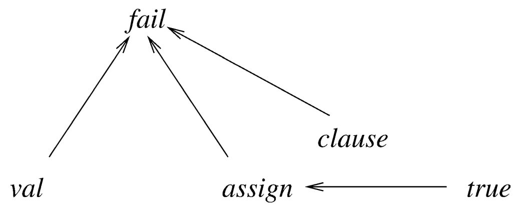


Fig. 3. Dependency graph for the specification of the 3-SAT problem.

important cases we can discard several instantiations of such rules. As an example, in rule (8) if `s` is a DB predicate, then we have to consider only value assignments to `Y`, `Z` and `W` which correspond to facts in *DB*, instead of all $|U|^3$ assignments.

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

- if p is a primitive predicate, $\mathit{alive}(p)$ is the set of the ground instantiations corresponding to variables of the kinds α and β;
- if p is a defined predicate, $\mathit{alive}(p)$ 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; the topological sort is not unique, but any sort ensures that all predicates in the body of a rule are processed before the predicate in the head, and therefore the algorithm works in the same way for any of them. The choice of the order in which nodes are visited may not affect the performance of the algorithm, since for processing a predicate occurring in the head of some rule all predicates occurring in the body must already have been evaluated. When a predicate p is under analysis, the set $\mathit{alive}(p)$ is built, and clauses (kind `c`) corresponding to instantiations of rules with p in the head are generated. Referring to the above example, in rule (8) we must consider only value assignments to the four NP-SPEC variables `X`, `Y`, `W`, and `Z`, according to $\mathit{alive}(q)$, $\mathit{alive}(s)$, and $\mathit{alive}(r)$.

3.2.2. Partial rule instantiation

The above technique reduces the size of the generated propositional formula T. We now describe an optimization which does not reduce the size of the formula, but rather generates it in a shorter time.

This technique concerns the way assignments to the NP-SPEC variables occurring in a rule of a specification are generated. As we previously mentioned, the number n of distinct variables in a rule is a crucial parameter, because in the worst case $|U|^n$ variable assignments must be taken into account during the process of variable instantiation. In very simple specifications, n can be as large as 10 (cf. the first rule in the specification of job shop scheduling in Section 4.3). The size $|U|$ of the Herbrand universe depends on the instance, and a size greater than 100 is very common. As a consequence, 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 in which n distinct variables occur, explores a tree of depth n, and uses sets *alive* for pruning the search. At each step the algorithm considers a *partial* assignment, i.e., an assignment to a subset of m variables, with $m < n$. For example, referring to rule (8), if `Z` and `W` have already been assigned to, e.g., 2 and 7, and $\mathtt{r}(2, 7) \notin \mathit{alive}(\mathtt{r})$, it is useless to consider assignments to the other variables `X` and `Y`.

In addition, relational operators are evaluated as soon as their operands are instantiated, and the algorithm backtracks as soon as an expression involving a relational operator is evaluated to *false*, since the body of the rule is false.

3.2.3. *Inlining predicates*

A further idea that dramatically improved the performance of the compiler concerns a special treatment (that we named "inlining") for some of the predicates, which is done as the first step. Potentially, a predicate can be inlined if we know exactly all its possible instantiations; this happens, for example, for predicates in *DB*, and for those which are defined using only *DB* and relational operators. These are "fixed-extension" predicates, since their extension can be evaluated without resorting to guessed predicates. We call such predicates *inlineable*; an example of an inlineable predicate is the predicate `val` in the specification of 3-SAT in Section 2.1. From here on, we denote the extension of an inlineable predicate with $ext(P)$.

The inlining of a predicate d works as follows: each positive occurrence of d is instantiated in all possible ways, instantiating the corresponding variables. The corresponding rule is thus replicated for all possible instantiations. Moreover, if $d \in DB$ occurs always positively in all bodies, then it can be completely eliminated from *DB*.

As an example, consider the following specification fragment, in which `p` is a *DB* predicate.

```
DATABASE
  p = {(1,2,3),(2,4,5),(4,6,7)};
  ...
SPECIFICATION
  r(X,W,T) <-- p(X,Y,Z), p(Y,V,W), q(Z,V,T).  // R1
  ....
```

If `p` is declared "inline", this rule is replaced by the set of rules obtained by all possible instantiation of `p`. In a first step of the inlining process, we replace the first occurrence of the literal `p`, yielding to the following set of rules (there are three rules because the first occurrence of `p` can be unified in three different ways with atoms of *DB*):

```
  r(1,W,T) <-- p(1,2,3), p(2,V,W), q(3,V,T).
  r(2,W,T) <-- p(2,4,5), p(4,V,W), q(5,V,T).
  r(4,W,T) <-- p(4,6,7), p(6,V,W), q(7,V,T).
```

In the second step, we replace also the literal `p(Y,V,W)`, thus obtaining the following two rules (there are two of them because the second occurrence of `p` can be unified in two different ways with atoms of *DB*):

```
  r(1,5,T) <-- p(1,2,3), p(2,4,5), q(3,4,T).  // R2
  r(2,7,T) <-- p(2,4,5), p(4,6,7), q(5,6,T).  // R3
```

The occurrences of `p` in `R2` and `R3` are then removed. In addition, if `p` does not appear negatively in any rule of *SP*, at the end of the inlining process the predicate `p` is removed. The resulting specification is the following:

```
DATABASE
  ... // p removed
```

```
SPECIFICATION
  r(1,5,T) <-- q(3,4,T).  // R4
  r(2,7,T) <-- q(5,6,T).  // R5
  ....
```

The order upon which the literals are inlined may affect the efficiency of the translation. In order to keep the number of intermediate rules as small as possible, we first inline the literals whose predicate has the smallest extension.

Although the inlining process in general increases the number of rules in the specification, there is a beneficial effect on the number of distinct NP-SPEC variables in a rule (which, as discussed before, is a crucial parameter), because several of the variables are eliminated. As an example, in the job-shop scheduling specification in Section 4.3, 8 (including the 3 mute ones) out of 10 variables in the first rule are eliminated by inlining the predicate `task`.

3.2.4. The complete algorithm

We end this section by giving an outline of the algorithm of SPEC2SAT which accounts for the features we have described. The module SPEC2SAT corresponds to about 2500 lines of C++ code. For the sake of readability, we have modularized the algorithm by defining three auxiliary procedures.

```
Algorithm SPEC2SAT
Input a specification $S = \langle DB, SP\rangle$ without aggregates, recursion, and
negative occurrences of defined predicates
Output a CNF formula $T$ 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 $S$
begin
   // step 0: preliminary definitions
   set of clauses $T = \emptyset$;
   int $n$ = number of predicate symbols occurring in $S$;
   array of subsets of the Herbrand base $alive[n]$;
   $G$ = the dependency graph of $SP$;
   $TS$ = any topological sort of $G$;
   for each predicate symbol $p$ occurring in $S$
      $alive[p] = \emptyset$;
   // step 1: perform inlining
   $S$ = PerformInlining($S$);
   // step 2: deal with primitive predicates
   $\langle T, alive[\,]\rangle$ = ProcessPrimitivePredicates($S, T, alive[\,]$);
   // step 3: deal with defined predicates
   $T$ = ProcessDefinedPredicates($S, T, alive[\,]$);
   return $T$;
end. // SPEC2SAT
```

```
Procedure PerformInlining
Input a specification $S = \langle DB, SP\rangle$ (restrictions of SPEC2SAT apply)
Output the modified specification $S$, equivalent to the input one
begin
  set of predicates $I$ = inlineable predicates of $SP$;
  for each predicate $d \in I$ (following the order of $TS$)
  do begin
     for each rule $r \in SP$ in which $d$ occurs
     do begin
        for each fact $d(a)$ s.t. $a \in ext(d)$
        do begin
           $r' =$ the instantiation of $r$ with $a$;
           $SP = SP \cup \{r'\}$;
        end; // for each fact $d(a)$ s.t. $a \in ext(d)$
        $SP = SP \setminus \{r\}$;
     end; // for each rule $r \in SP$ in which $d$ occurs
     if $d$ occurs always positively in the rules of $SP$
     then eliminate $d$ from $SP$;
  end; // for each predicate $d \in I$
  return $S$;
end; // PerformInlining
```

```
Procedure ProcessPrimitivePredicates
Input a specification $S = \langle DB, SP\rangle$ (restrictions of SPEC2SAT apply)
a set of clauses $T$, an array of subsets of the Herbrand base $alive[\,]$
Output the modified pair $\langle T, alive[\,]\rangle$, encoding semantics of primitive predicates
begin
  for each predicate $d \in DB$
  do begin
     $alive[d] = \{d(a) \mid d(a) \in DB\}$;
     $T = T \cup alive[d]$;
  end; // for each predicate $d \in DB$
  for each guessed predicate $q$ defined in metaclause $m$ of $SP$
  do begin
     $alive[q] =$ all ground instantiations defined by $m$;
     if $m$ declares a Partition
     then $T = T \cup$ the propositional encoding of $m$
                 defined by clauses of the form (1)–(2);
     else if $m$ declares a Permutation
     then $T = T \cup$ the propositional encoding of $m$
                 defined by clauses of the form (3)–(5);
     else if $m$ declares an IntFunc
     then $T = T \cup$ the propositional encoding of $m$
                 defined by clauses of the form (6)–(7);
     end; // if $m$ declares
  end; // for each guessed predicate $q$ defined in metaclause $m$ of $SP$
return $\langle T, alive[\,]\rangle$;
end; // ProcessPrimitivePredicates

Procedure ProcessDefinedPredicates
Input a specification $S = \langle DB, SP\rangle$ (restrictions of SPEC2SAT apply)
a set of clauses $T$, an array of subsets of the Herbrand base $alive[\,]$
Output the modified set of clauses $T$, encoding semantics of defined predicates
begin
  for each defined predicate $p$ (following the order of $TS$)
  do begin
      for each rule $r$ with $p$ in the head
      do begin
          for each instantiation $\sigma$ of the variables occurring in $r$
          // actually, partial instantiations are considered,
          // with a backtracking-like algorithm
          do begin
              $r'$ = the instantiation of $r$ with $\sigma$;
              $p(w)$ = the head of $r'$;
              if there is not an atom $a(t)$ in the body of $r'$ such that $t \notin alive[a]$
              then begin
                  $alive[p] = alive[p] \cup \{w\}$;
                  $T = T \cup r'$;
              end;
          end; // for each instantiation $\sigma$ of the variables occurring in $r$
      end; // for each rule $r$ with $p$ in the head
  end; // for each defined predicate $p$ (following the order of $TS$)
  return $T$;
end; // ProcessDefinedPredicates
```

4. Performance considerations

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 performance, our system cannot outperform 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 an efficient solver for a new problem in NP may take weeks or even months.

Nevertheless, we wish to show that the system is able to solve medium-size instances of various classical hard problems. Conversely, the algorithm of the original PROLOG engine of NP-SPEC is able to solve only small instances. We believe that the ability to solve non-trivial cases helps the user get a better understanding of her/his application and to capture aspects of the problem that may fail to appear in very small instances.

In the following subsections, we analyze the performance of our system on four problems: graph coloring, Hamiltonian cycle, job shop scheduling, and tournament scheduling. Experiments use the solver SATZ and were run on an AMD Athlon 1,533 MHz PC running Linux, with the GNU `g++` compiler (version 2.96). Times are expressed in seconds of CPU use.

The total time for finding a solution is the sum of the compilation time t_1 and the time t_2 needed by the SAT solver 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 $t_1 > t_2$, such as 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 $n \cdot k$ propositional variables, one for each pair $(\langle node \rangle, \langle color \rangle)$. The formula contains $O(n \cdot k^2)$ clauses of kind `b` which state that each node has exactly one color. The rule `GC2` adds $e \cdot k$ clauses of kind `c` that forbid two adjacent nodes to have the same color (cf. Fig. 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 [24]. Table 1 reports our results.

The table shows that the system has been able to solve some large instances. In one case, the instance DSJC.125.1, it has also been able to prove the minimality of $k = 5$. This result was obtained by proving the unsatisfiability of the formula generated with 4 colors (first line of the table). Conversely, in 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.

Table 1
Performance on the graph coloring problem

Graph name	Nodes	Edges	Colors (min)	Colorable	Compile time	SAT time	Variables	Clauses
125.1	125	736	4	NO	0.7	0.08	500	3,819
125.1	125	736	5 (5)	YES	0.33	0.05	625	5,055
125.5	125	7,782	21 (16)	YES	6.53	16.16	2,625	108,086
250.1	250	3,218	9 (9)	YES	7.49	1.34	2,250	41,430
250.5	250	31,336	39 (27)	YES	58.56	64.76	9,750	796,552
500.1	500	12,456	16 (14)	YES	57.82	23.03	8,000	272,286
1000.1	1,000	49,629	26 (19)	YES	459.12	6.63	26,000	1,665,983

Table 2
Performance on the Hamiltonian cycle problem

Nodes	Edges	Cycle	Compile time	Avg. SAT time	Variables	Clauses
15	30	NO	0.22	131.33	225	6,090
15	60	NO	0.22	–	225	5,640
15	60	YES	0.22	0.24	225	5,640
15	90	YES	0.22	0.09	225	5,190
17	35	NO	0.30	–	289	8,976
17	70	NO	0.30	–	289	8,364
17	70	YES	0.30	1.02	289	8,364
17	105	YES	0.30	0.14	289	7,752
20	43	NO	0.51	–	400	14,780
20	86	NO	0.51	–	400	13,900
20	86	YES	0.51	–	400	13,900
20	130	YES	0.51	76.82	400	13,020

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 propositional 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 $O(n^3)$ (cf. set of clauses (3)–(5)). 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 tested our system on random instances. It is known [25] that the most difficult random instances are obtained for a number of edges e equal to $p = n \log n/2$, i.e., p is the so-called "crossover point" in which half of the instances have no solution. 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). The symbol "–" means that SATZ did not terminate within half an hour in at least one instance. We note that compilation is quite fast, while the SAT solver is very slow. In fact, the solver is able to easily handle 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 totally ineffective for unsatisfiable ones.

Experiments with other SAT solvers provide similar results, whereas current solvers of the Hamiltonian cycle problem are able to solve larger instances quite easily (cf., e.g., [26]). Therefore, we conclude that the translation into SAT using our encoding is not effective for this problem.

4.3. Job shop scheduling

Job shop scheduling [2, 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 an integer-valued 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
   nb_tasks = 36;  D = 55; // instance FT06
      // task(T,J,Po,Pr,L): the task T belongs to job J in
      // position Po, and 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..nb_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.
```

Compilation of this NP-SPEC file generates a SAT instance with $m \cdot D$ propositional 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), both available from the OR library at http://www.ms.ic.ac.uk/info.html.

As shown in Table 3 the first instance is solved easily, and the proof that the deadline is optimal (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 $D = 43$, which corresponds to 860 ($= 43 \cdot 20$) in LA02. This result took quite a long time; but, if we give looser deadlines, i.e., $D = 46, 48$, and 50, the problem is solved quicker. We remark that the

Table 3
Performance on the job shop scheduling problem

Instance	Tasks	Deadline	Solvable	Compile time	SAT time	Variables	Clauses
FT06	36	54	NO	7.53	17.28	1,944	203,792
FT06	36	55	YES	7.89	4.57	1,980	214,034
LA02	50	860	YES	9.02	732.17	2,150	201,813
LA02	50	920	YES	9.34	119.98	2,300	228,929
LA02	50	960	YES	9.87	23.26	2,400	236,508
LA02	50	1,000	YES	10.27	3.60	2,500	257,745

minimum value of the deadline for this instance is 655, and all state-of-the-art solvers find solutions below 700.

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

We remark that, without the optimization steps described in Section 3.2, none of the instances of Table 3 was compiled by SPEC2SAT in less than 24 hours.

4.4. Tournament scheduling

As the final example, we propose a real world problem whose specification is much more complex than the classical academic problems proposed so far.

The *Tournament Scheduling* problem consists in assigning the matches to rounds of a round-robin tournament for a sport league. Many different versions of this problem have been proposed in the literature. We specify here the version solved in [27]. However, given that the problem in [27] is an optimization problem, we omit the objective function and focus on the underlying search problem.

We are given $2n$ teams and a tournament *pattern*, which is a complete round-robin tournament in which numbers from 1 to $2n$ are used as teams. An example of a pattern for $2n = 6$ is given in Fig. 4, where the order of the teams determines the location of the match: The first team plays home and the second away.

The problem consists in finding a matching between the actual teams and the numbers, called *dummy teams*, appearing in the pattern. The matching has to satisfy the following general constraints:

Round 1	1-6	2-5	4-3
Round 2	6-2	3-1	5-4
Round 3	3-6	4-2	1-5
Round 4	6-4	5-3	2-1
Round 5	5-6	1-4	3-2

Fig. 4. A pattern for $2n = 6$.

Complementarity: Teams t_1 and t_2 must have *complementary schedules*: For each round r, if t_1 plays home in r then t_2 plays away in r, and if t_1 plays away in r then t_2 plays home in r.
Availability: Team t must play away at round r.
NoMating: Team t_1 cannot play home with team t_2 at round r.
Triplets: Three teams t_1, t_2, and t_3 cannot be simultaneously home in the same round.

Besides the above general ones, there is a second group of constraints which presupposes some prior notions: We call *top teams* the members of a subset of the teams composed by the strongest teams, which require special treatment. We call *top match* a match between two top teams. We call *distance* of two matches the number of rounds between those in which they take place. The constraints are the following:

Top matches schedule: For a given set of rounds R, no top match can take place at any round $r \in R$.
Top matches distance: Two top matches cannot take place at distance smaller than a given value *TopMatchDistance*.
Top opponent distance: Any team cannot match two top teams at distance smaller than a given value *TopOpponentDistance*.

A sample instance in NP-SPEC is the following.

```
DATABASE
  nb_teams = 6;
  top_match_distance = 1;
  top_opponent_distance = 1;

  // pattern(R,D1,D2): D1 and D2 play at round R at D1'S home
  // stadium
  pattern = {(1,1,6), (1,2,5), (1,4,3),
             (2,6,2), (2,3,1), (2,5,4),
             (3,3,6), (3,4,2), (3,1,5),
             (4,6,4), (4,5,3), (4,2,1),
             (5,5,6), (5,1,4), (5,3,2)};
  // no_mating(R,T1,T2): T1 and T2 cannot play at round R at
  // T1's home
  no_mating = {(1,1,2), (3,4,6)};
  // unavailable(R,T1): T1 cannot play home at round R
  unavailable = {(1,1),(3,5)};
  // triplet(T1,T2,T3): T1, T2 and T3 cannot be home together
  triplet = {(1,4,6)};
  complementary_teams = {(3,4)};
  top_teams = {(1),(2)};
  forbidden_rounds_for_top_matches = {(1),(5)}.
```

The specification in NP-SPEC is the following. For the sake of readability, variables denoting dummy teams are named `D`, `D1`, `D2`, ..., variables denoting actual teams are named `T`, `T1`, `T2`, ..., and variables denoting rounds are named `R`, `R1`, `R2`, ...

```
SPECIFICATION
       // assignment(D,T): D is assigned to T
    Permutation({1..nb_teams}, assignment).

       // rule 1: mating constraints
    fail <-- no_mating(R,T1,T2), assignment(D1,T1),
             assignment(D2,T2), pattern(R,D1,D2).

       // rule 2: availability constraints
    fail <-- unavailable(R,T1), assignment(D1,T1),
             pattern(R,D1,_).

       // rule 3: triplet constraints
    fail <-- triplet(T1,T2,T3),
             assignment(D1,T1), assignment(D2,T2),
             assignment(D3,T3), pattern(R,D1,_), pattern(R,D2,_),
             pattern(R,D3,_).

       // rules 4 and 5: definition of non-complementary pairs of
    dummy teams overlapping_dummy_teams(D1,D2) <-- pattern(R,D1,_),
    pattern(R,D2,_).
    overlapping_dummy_teams(D1,D2) <-- pattern(R,_,D1),
    pattern(R,_,D2).

       // rule 6: complementarity constraints
    fail <-- complementary_teams(T1,T2),
             assignment(D1,T1), assignment(D2,T2),
             overlapping_dummy_teams(D1,D2).

       // rules 7 and 8: definition of separation between rounds
    separation(R1,R2,D) <-- R1 > R2, R1 - R2 > D.
    separation(R1,R2,D) <-- R2 > R1, R2 - R1 > D.

       // rule 9: top matches distance constraints
    fail <-- top_teams(T1), top_teams(T2), top_teams(T3),
             top_teams(T4),
             T1 * nb_teams + T2 != T3 * nb_teams + T4,
             T2 * nb_teams + T1 != T3 * nb_teams + T4,
             assignment(D1,T1), assignment(D2,T2),
             assignment(D3,T3), assignment(D4,T4),
             pattern(R1,D1,D2), pattern(R2,D3,D4),
             separation(R1,R2,top_match_distance).

       // rules 10 and 11: definition of game between dummy teams,
       //                  regardless of the order
```

```
game(R,D1,D2) <-- pattern(R,D1,D2).
game(R,D1,D2) <-- pattern(R,D2,D1).

   // rule 12: top matches schedule constraints
fail <-- top_teams(T1), top_teams(T2),
         forbidden_rounds_for_top_matches(R),
         assignment(D1,T1), assignment(D2,T2),
         game(R,D1,D2).

  // rule 13: top opponent distance constraints
fail <-- top_teams(T1), top_teams(T2), NOT top_teams(T3),
         T1 < T2, assignment(D1,T1), assignment(D2,T2),
         assignment(D3,T3),
         game(R1,D1,D3), game(R2,D2,D3),
         separation(R1,R2,top_opponent_distance).
```

In this specification we make extensive use of defined predicates. In particular, we have `overlapping_dummy_teams` representing the fact that two dummy teams are not complementary, `separation` representing minimum distance between rounds, and `game` representing matches between dummy teams no matter of the order.

Notice in rule 9 the two conditions `T1 * nb_teams + T2 != T3 * nb_teams + T4` and `T2 * nb_teams + T1 != T3 * nb_teams + T4`, which state that the unordered pair (`T1`,`T2`) is different from the unordered pair (`T3`,`T4`).

Note also the condition `T1 < T2` in rule 13. This is a small concession to efficiency, because the looser condition `T1 != T2` would be correct as well but would generates duplicate clauses.

We experiment with the same real instances used in [27] with $2n = 18$, but we perturb some of the data in order to understand the behavior of the translation in different conditions. The most important parameters turned out to be (not surprisingly) the number of top teams and the two values `top_match_distance` and `top_opponent_distance`.

Table 4 shows the results for various values of these parameters, while keeping the other data to their normal values. The symbol "–" means that SATZ did not terminate within half an hour.

The specialized solver in [27] is also sensitive to these parameters, and takes between 3 seconds and 3 minutes to solve these types of instances.

Table 4
Performance on the tournament scheduling problem for $2n = 18$

Number of top teams	Top match distance	Top opponent distance	Compile time	SAT time	Variables	Clauses
0	–	–	2.33	0.34	324	18,315
2	1	1	2.48	0.32	324	19,035
2	1	2	7.14	–	324	33,179
3	1	1	3.22	0.38	324	19,935
3	1	2	9.88	–	324	59,715

Table 4 shows that we can rapidly solve instances with few or no special constraints, whereas a large number of them make `satz` unable to find a solution.

This result somewhat confirms that problems involving permutations, like the Hamiltonian cycle, are difficult to solve with `satz`.

4.5. Further considerations

As a final comment on this analysis, we notice that compilation and execution performance are completely independent of each other, and can be very different from problem to problem. In some cases, e.g., Hamiltonian path, compilation is much faster than solving the SAT instance, while in others, e.g., graph coloring, they are comparable.

The cost of compilation depends mostly on the number of rules, the number of variables in each rule, and the size of the Herbrand universe. Conversely, the cost of the solver depends on the combinatorial structure of the instance.

Obviously, the specification can play an important role too. For example, job shop scheduling could be specified in a different way based on permutations of tasks on every processor. Such an alternative specification would result in different performance numbers.

In order to understand the overhead introduced by SPEC2SAT, we conclude this section by briefly analyzing its behavior for the translation of the 3-SAT problem (cf. Section 2.1). We thus compare the running times with the sizes of the input and the output formulas.

Recalling the specification of Section 2.1, we call n and c the number of propositional variables and the number of clauses of the input formula in the `DATABASE` section. We first realize that the output formula has $3n$ variables, namely n for the predicate `true` and $2n$ for the predicate `assign`. No variable is created for the predicate `val` because this is "inlined", and thus removed.

The number of clauses is $2n + c$, given that $2n$ of them are generated to describe the behavior of the predicates `true` and `assign`, i.e., either `true(l)` or `NOT true(l)` must be true, and the others are clauses in one-to-one correspondence with the clauses of the original formula.

We have thus shown that the overhead in terms of space is linear in the number of variables.

Experiments show that the running times of `satz` for solving the input and the output formulas are also very close. For example, for instances with $n = 260$ and $c = 1{,}118$, the running time of the output formula is in the average 2.5 times higher than the running time of the input formula. A more analytical comparison is not possible at this stage, given that we use the solver as a "black box".

5. Related work

Other researchers have proposed DATALOG-like languages for problem specification. The main difference between NP-SPEC and other such languages—notably, DLV [28] and SMODELS [29]—is 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 [30]. The main difference with respect to 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.

Regarding the encoding, in this work we have used a general mechanism that creates a propositional variable for every element of the Herbrand base. This choice ensures the required generality for expressing all kind of problems. However, for some specific structures of the search space it is often possible to resort to ad hoc encodings that are more likely to allow efficient search strategies in the average case. We want to discuss the differences of our approach with respect to "manual" translations on some of the problems that we have considered in Section 4.

Crawford and Baker [7] propose an ad hoc SAT encoding for a generalization of the job shop scheduling problem (cf. Section 4.3), the so-called *machine shop scheduling* problem (see [31]). The generalization consists in task-ordering and machine-capacity constraints having a general form, whereas in job shop they have the fixed form given by job structure (chains of tasks) and by one-job-at-the-time behavior of the machines. The encoding of Crawford and Baker creates $2mp$ variables, two for each pair (t, i), one that represents the fact that t starts at i or later and one that represents the fact that t ends at i or earlier. In addition, they include a set of $m \cdot (m + 1)/2$ variables, called *precedence variables* that for each pair of tasks (t_1, t_2) state which of the two is processed first. Crawford and Baker discuss the encoding generated by SPEC2SAT, which creates one variable for each pair task/time point, called the "obvious" one, and they claim that their encoding is more efficient. This is probably due to the presence of the precedence variables that guide search toward feasible assignments, however the claim is neither explained nor supported in their paper.

Iwama and Miyazaki [32] propose a specific SAT translation for Hamiltonian cycle (cf. Section 4.2). Unlike the SPEC2SAT algorithm, their translation is based on a binary encoding of integer variables. Therefore, they produce a formula with a number of variables that is logarithmic with respect to the one produced by Spec2Sat.

The Iwama–Miyazaki translation is used by Hoos [33] to study the performance of the solver GSAT on instances obtained by translation from Hamiltonian cycle. Hoos shows that such instances are generally difficult for GSAT, and he provides various explanations for it. Going into the details of this phenomenon is beyond the scope of this paper, however, our results also confirm that instances obtained from Hamiltonian cycle are difficult not only for SATZ, as noticed in Section 4.2, but also for other SAT solvers.

The Hamiltonian cycle belongs to the class of problems called *permutation problems* in the context of constraint satisfaction problems (CSP). Roughly speaking, a CSP permutation problem is a problem that has the same number of variables and domain values, and in which it is possible to exchange variables and values (see, e.g., [34]). In the CSP community there has been quite a large body of research on permutation problems. It has been shown that permutation problems can be reformulated in a *dual* form, and such formulation can be used in alternative or in combination with the *primal* (i.e., the original one). Different (but equivalent) combinations of variables and constraints generate representations of the problem that can have computational properties that differ significantly from one another. The CSP formulation can be translated into SAT in various ways. The main point in the translation regards the number of propositional letters used to encode each CSP variable. Such numbers can be either linear or logarithmic in the domain size (see

[35]), corresponding to a unary or binary encoding of the domain. An example of binary encoding is the Iwama-Miyazaki translation mentioned above. Thus combining the representations of permutation problems in CSP with the different ways to translate a CSP into SAT, we obtain a large variety of possible options, at least for permutation problems. The automatic detection of situations in which an alternative encoding could be used would very likely provide a large improvement for our system, and will be addressed in a future project.

6. Conclusions and future work

We have presented a novel approach for the execution of specifications of problems in NP, based on translation into SAT. The performance of the resulting system is very good, compared to the previous, PROLOG-based, engine underlying NP-SPEC. For example, we were able to solve benchmarks of graph coloring problems with 1,000 nodes, while the previous approach was able to deal with graphs of just 14 nodes. As another example, we were able to increase the size of the chessboard in the n-queens problem 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 lead to improvements 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 [20]. Using SPEC2SAT it is possible to generate SAT instances from any NP-complete problem.

We claim that SPEC2SAT is one of the few research projects that aim to provide a highly declarative language for constraint solving that is capable of: 1) specifying all problems in a significant complexity class, and 2) solving non-trivial instances. The distinguishing feature of SPEC2SAT with respect to other projects such as DLV or SMODELS is the capability of using any third-party solver for SAT as a black box.

In the future, we plan to include aspects of NP-SPEC that have been neglected in this version, such as aggregates and recursion. As far as the efficiency is concerned, we plan to translate the specification to improve compilation in terms of both the size and the difficulty of the generated formula. For example, as already mentioned, the management of permutation problems could be significantly improved in various ways.

As shown in Section 4, sometimes the bottleneck is just the size of the SAT instance; therefore languages more expressive than propositional logic could be used so as to have a smaller output from the compilation phase. As an example, we could use quantified Boolean formulae as the target language, for which efficient solvers exist (cf., e.g., [36]).

Finally, we wish to equip the system with a learning mechanism for automatic selection of the best SAT solver for the instance at hand. For example, 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, and to Adrian Rollett for careful proofreading. Anonymous referees' comments and suggestions led to substantial improvements in the quality of this paper.

Appendix A. Syntax of NP-SPEC

NP-SPEC is case-sensitive, and the syntax of the full language is as follows.

```
<upper_case_letter> ::= A | ... | Z
<lower_case_letter> ::= a | ... | z
<letter>            ::= <upper_case_letter> | <lower_case_letter>
<string>            ::= <letter> | <string> <letter>

<variable_id>   ::= <upper_case_letter>
                    | <upper_case_letter> <string>
<symbol>        ::= <lower_case_letter>
                    | <lower_case_letter> <string>
<predicate_id>  ::= <lower_case_letter>
                    | <lower_case_letter> <string>
<domain_id>     ::= <string>
<constant_id>   ::= <string>

<digit>   ::= 0 | ... | 9
<digits>  ::= <digit> | <digits> <digit>
<integer> ::= <digits> | - <digits>

<spec_program>  ::=  <instance> <specification>
<instance>      ::=  DATABASE <declarations>
<declarations>  ::=  <declaration> | <declarations> <declaration>
<declaration>   ::=  <domain_id> = { <extension> } ;
                   | <constant_id> = <integer> ;
<extension>     ::=   <tuples> | <interval>
<tuples>        ::=   <tuple> | <tuples> , <tuple>
<tuple>         ::=   ( <sequence> ) | <value>
<sequence>      ::=   <value> | <sequence> , <value>
<value>         ::=   <integer> | <symbol>
<interval>      ::=   <integer_constant>..<integer_constant>
<integer_constant> ::= <integer> | <constant_id>

<specification>  ::=  SPECIFICATION <metapredicates> <rules>
<metapredicates> ::=  <metafact> | <metapredicates> <metafact>
<metafact> ::=  Subset ( <domain> , <predicate_id> ) .
              | Permutation ( <domain> , <predicate_id> ) .
              | Partition ( <domain> , <predicate_id>,
                             <integer_constant> ) .
                 | IntFunc ( <domain> , <predicate_id> ,
                           <interval> ) .

<domain>        ::=  <domain_id>
                  |  { <extension> }
                  |  <domain> + <domain>
                  |  <domain> * <domain>
                  |  <domain> - <domain>
                  |  <domain> >< <domain>

<rules>         ::=  <rule> | <rules> <rule>
<rule>          ::=  <atom> <-- <body> .

<atom>          ::=  <predicate_id>
                  |  <predicate_id> ( <terms> )

<terms>         ::=  <term> | <terms> , <term>
<term>          ::=  <integer_constant> | <value>
                     | <variable_id> | _

<body>          ::=  <literal> | <body> , <literal>
<literal>       ::=  <atom> | NOT <atom> |
                   | <expression> <relational_operator> <expression>
                   | <aggregate>

<expression>    ::=  <integer_constant>
                   | <variable_id>
                   | <expression> + <expression>
                   | <expression> - <expression>
                   | <expression> * <expression>
                   | <expression> / <expression>
                   | ( <expression> )

<relational_operator> ::=  > | < | >= | <= | == | <>

<aggregate>     ::= <agg_op> ( <predicate_id> ( <agg_terms> ) ,
                              <variable_id> )
<agg_terms>     ::= <agg_terms> , <agg_term> | <agg_term>
<agg_term>      ::= * | <term>
<agg_op>        ::= COUNT | MAX | MIN | SUM
```

References

[1] M.H. van Emden, R.A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23 (4) (1976) 733–742.

[2] M.R. Garey, D.S. Johnson, Computers and Intractability—A Guide to NP-Completeness, W.H. Freeman, San Francisco, CA, 1979.

[3] D.S. Johnson, M.A. Trick (Eds.), Cliques, Coloring, and Satisfiability. Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, American Mathematical Society, Providence, RI, 1996.

[4] H.A. Kautz, B. Selman, Planning as satisfiability, in: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), 1992, pp. 359–363.

[5] H.A. Kautz, B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), Portland, OR, 1996, pp. 1194–1201.

[6] H.A. Kautz, D. McAllester, B. Selman, Encoding plans in propositional logic, in: Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96), 1996, pp. 374–384.

[7] J.M. Crawford, 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), Seattle, WA, 1994, pp. 1092–1097.

[8] M. Fujita, J. Slaney, F. Bennett, Automatic generation of some results in finite algebra, in: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, 1993, pp. 52–57.

[9] T. Larrabee, Test pattern generation using Boolean satisfiability, IEEE Trans. Computer-Aided Design (1992) 4–15.

[10] F. Massacci, L. Marraro, Logical cryptanalysis as a SAT-problem: encoding and analysis of the US Data Encryption Standard, J Automat. Reason. 24 (1–2) (2000) 165–203.

[11] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, D. Vasile, NP-SPEC: an executable specification language for solving all problems in NP, Computer Languages 26 (2000) 165–195.

[12] K.R. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 89–142.

[13] V. Lifschitz, Computing circumscription, in: Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85), Los Angeles, CA, 1985, pp. 121–127.

[14] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: R.M. Karp (Ed.), Complexity of Computation, American Mathematical Society, Providence, RI, 1974, pp. 43–74.

[15] A. Aggoun, et al., ECL^iPS^e User Manual (Version 4.0), IC-Parc, London, UK, July 1998.

[16] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.

[17] M. Davis, G. Logemann, D.W. Loveland, A machine program for theorem proving, Comm. ACM 5 (7) (1962) 394–397.

[18] B. Selman, D. Mitchell, H. Levesque, Generating hard satisfiability problems, Artificial Intelligence 81 (1996) 17–29.

[19] B. Selman, H. Kautz, B. Cohen, Noise strategies for improving local search, in: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), Seattle, WA, 1994, pp. 337–343.

[20] SATLIB, The Satisfiability Library, http://www.satlib.org.

[21] C. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, 1997, pp. 366–371.

[22] M. Cadoli, L. Palopoli, Circumscribing DATALOG: expressive power and complexity, Theoret. Comput. Sci. 193 (1998) 215–244.

[23] E. Horowitz, S. Sahni, Fundamentals of Data Structures, Pitman, London, 1976.

[24] D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning, Oper. Res. 39 (3) (1991) 378–406.

[25] P. Cheeseman, B. Kanefski, W.M. Taylor, Where the really hard problem are, in: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney, Australia, 1991, pp. 163–169.

[26] B. Vandegriend, Hamiltonian cycles: algorithms, graphs and performance, MSc Thesis, University of Alberta, Edmonton, AB, February 1998.

[27] A. Schaerf, Scheduling sport tournaments using constraint logic programming, CONSTRAINTS 4 (1) (1999) 43–65.

[28] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, The KR system `dlv`: progress report, comparisons and benchmarks, in: Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, 1998, pp. 406–417.

[29] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artificial Intelligence 25 (3–4) (1999) 241–273.

[30] D. Jackson, Automating first-order relational logic, in: Proc. of ACM SIGSOFT'00: 8th SIGSOFT Symposium on Foundations of Software Engineering, 2000, pp. 130–139.

[31] N. Sadeh, Look-ahead techniques for micro-opportunistic job shop scheduling, Tech. Rept. CMU-CS-91-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, available at http://www.cs.cmu.edu/~sadeh, 1992.

[32] K. Iwana, S. Miyazaki, SAT-variable complexity of hard combinatorial problems, in: Proc. of the World Computer Congress of the IFIP, vol. 1, Elsevier Science, Amsterdam, 1994, pp. 253–258.

[33] H.H. Hoos, Solving hard combinatorial problems with GSAT—a case study, in: Proceedings of the Twentieth German Annual Conference on Artificial Intelligence (KI'96), in: Lecture Notes in Computer Science, vol. 1138, Springer, Berlin, 1996, pp. 197–212.

[34] B.M. Smith, Dual model of permutation problems, in: Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001), in: Lecture Notes in Computer Science, vol. 2239, Springer, Berlin, 2001, pp. 615–619.

[35] T. Walsh, Permutation problems and channelling constraints, in: Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001), Springer, Berlin, 2001, pp. 377–391.

[36] M. Cadoli, M. Schaerf, A. Giovanardi, M. Giovanardi, An algorithm to evaluate quantified boolean formulae and its experimental evaluation, J. Automat. Reason. 28 (2002) 101–142.