Technical report 13-00 (July 24, 2000). Revised version as of April 18, 2001. Dipartimento di Informatica e Sistemistica. Università di Roma "La Sapienza", ITALY.
To appear on Computer Languages*, Pergamon – Elsevier.*
ftp://ftp.dis.uniroma1.it/pub/ai/papers/cado-etal-00-d-REVISED.ps.gz

NP-SPEC: An Executable Specification Language for Solving All Problems in NP*

Marco Cadoli¶, Giovambattista Ianni†, Luigi Palopoli#,
Andrea Schaerf‡, Domenico Vasile†
¶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

†Dipartimento di Elettronica, Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS), Italy
(ianni,vasile)@deis.unical.it
http://wwwinfo.deis.unical.it/~ianni

#Dip. di Informatica, Matematica, Elettronica e Trasporti
Università di Reggio Calabria,
Via Graziella, loc. Feo di Vito, I-89100 Reggio Calabria, Italy
palopoli@ing.unirc.it
http://www.ing.unirc.it/didattica/inform00/palopoli

‡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

April 18, 2001

Abstract

In this paper a logic-based specification language, called NP-SPEC, is presented. The language is obtained by extending DATALOG through allowing a limited use of some second-order predicates of predefined form. NP-SPEC programs specify solutions to problems in a very abstract and concise way, and are executable. In the present prototype they are compiled to PROLOG code, which is run to construct outputs. Second-order predicates of suitable form allow to limit the size of search spaces in order to obtain reasonably efficient construction of problem solutions. NP-SPEC

*A preliminary version of this paper appeared as [1, 2].

expressive power is precisely characterized as to express exactly the problems in the class NP. The specification of several combinatorial problems in NP-SPEC is shown, and the efficiency of the generated programs is evaluated.

Keywords: specification languages, logic programming, datalog.

1 Introduction

The definition and the implementation of logic-based languages allowing for specifying complex problems have recently received much attention in the research community (cf., e.g., [3, 4, 5, 6, 7]). The main features of such languages are:

- they are highly declarative, and, as such, they are easier to use and simplify the error-prone process of writing algorithms;
- they are executable, so that the defined specification can be actually run and thus constitutes a rapid prototype.

General logic-based languages that, being Turing-complete, allow for an unrestricted specification capability are proposed in, e.g., [5, 6]. Clearly, such languages are fairly complex, and this may be an obstacle for their use (cf., e.g., [8], where the difficulties arising with handling formal specifications is pointed out as a main factor in the limited use of formal methods in software development).

In this paper we follow a different direction, and propose a language with limited expressiveness. In particular, we define NP-SPEC, which is a highly declarative executable language, which allows the user to specify exactly all problems which belong to the complexity class NP in a simple way. This restriction has two advantages:

- the language is simpler, and easier to learn;
- a more focused strategy in the search for a solution is possible, thus leading to more efficient programs.

Like in [3, 4], we opt for a DATALOG-like, i.e., PROLOG with no function symbols, syntax. Basically, logical formulae are rules, hence they are more restricted than in general predicate logic. The main difference between NP-SPEC and the other languages relies in its semantics, which is based on the notion of *model minimality*. The different semantics makes negation in the body of rules not necessary –though permitted– thus allowing for simpler specifications.

We show an example of an NP-SPEC specification, which helps us to highlight the main aspects of the language. The example concerns the famous "3-coloring" problem, which is well-known to be NP-complete (cf. e.g., [9, problem GT4, page 191]), and is formally defined as follows.

INSTANCE:	Graph $G = (N, E)$
QUESTION:	Is G 3-colorable, i.e., does there exist a function $col : N \to \{1, 2, 3\}$ such that $col(u) \neq col(v)$ whenever $\{u, v\} \in E$?

In NP-SPEC, the user can make the following declarations, which specify both an instance (in the `DATABASE` section) and the question (in the `SPECIFICATION` section). In this case, the instance is a graph with six nodes and seven edges, which is 3-colorable.

```
DATABASE
   node = {1..6};
   edge = {(1,2), (1,3), (2,3), (6,2), (6,5), (5,4), (3,5)};

SPECIFICATION
   Partition(node,coloring,3).
   fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).
```

In the current implementation of NP-SPEC, processing the above specification gives as output an ECL^iPS^e, i.e., PROLOG, program. Running the ECL^iPS^e program produces the following output, which represents the colors to be assigned to the nodes so that the input graph is 3-colored.

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

NP-SPEC is a DATALOG-like language. In particular:

- the clauses in the `DATABASE` section specify a set of *facts*, i.e., function-free ground atomic first-order formulas, that define the instance;
- rules in the `SPECIFICATION` section are universally quantified function-free definite clauses, that specify the solution of the problem.

The main difference between NP-SPEC and DATALOG is the possibility of using second-order logic in a restricted way. Referring to the above example, the clause `Partition(node,coloring,3)` has the following meaning:

- The predicate `coloring`, which does not occur in the database section, is implicitly declared to have arity 2, i.e., the arity of domain `node` plus 1.
- The extension of `coloring` can be any set of the following kind:

$$\{\langle n, c\rangle \mid n \text{ is in the extension of } \texttt{node},\ c \in \{1, 2, 3\}\}$$

The intuitive meaning of the `SPECIFICATION` section is: the instance has no solution if *for all* possible extensions of `coloring`, there are nodes `X` and `Y` such that:

- `X` and `Y` are connected by an edge (cf. atom `edge(X,Y)`), and

- `X` and `Y` are colored with the same color `C` (cf. atoms `coloring(X,C)`, `coloring(Y,C)`).

In other words, an extension of `coloring` which makes `fail` true does not correspond to a solution.

The execution of the specification is such that as soon as the program finds an extension of `coloring` that does not make `fail` true, it outputs such an extension, or, at the request of the user, all extensions of this kind.

More precisely, in general, a specification in NP-SPEC has the following structure:

- The `DATABASE` section is used to specify the instance of a problem. As for the semantics, the extension of the predicates which are declared here (called *extensional* predicates) is always as specified by the set of facts.
- The `SPECIFICATION` section is used to:
  - Declare predicates which are used to represent the search space of the problem (e.g., `coloring`). As for the semantics, the extension of such predicates can be any subset of the Cartesian product of the corresponding domains. For this reason, we call such predicates *guessed* predicates.
  - Declare constraints binding extensional and guessed predicates, and possibly other predicates (one of them being the predefined predicate `fail`).

The main distinctive features of NP-SPEC are the following:

- It has a completely defined semantics, which is based on an extension of DATALOG known as DATALOG^{CIRC}, defined in [10].
- It has a precise connotation in terms of expressive power and computational complexity. In particular, it can specify exactly all problems in the complexity class NP.
- It offers a set of metapredicates, called *tailoring predicates*, which guide the system in the synthesis of algorithms that operate on search spaces of limited size so as to be reasonably efficient.

The language NP-SPEC is meant to be simple to use and to propose intuitive and concise specifications. Our approach is meant to lift the abstraction level at which the problem must be specified for the system to be able to generate automatically the code. The user is allowed to take the decisions at such an abstract level, and her/his decisions have an impact in the generated program in a transparent way.

In the rest of the paper, we outline the main technical aspects of NP-SPEC: syntax, semantics, and computational properties. We also briefly address the current prototype and its performances, report more specification examples, and discuss related work.

2 Preliminaries

In this section we recall some notions about complexity and expressiveness that will be useful in the following. The interested reader is referred to [11, 12]. Also, we recall the definition of DATALOG^{CIRC} [10], the language upon which the semantics of NP-SPEC relies.

2.1 Complexity and expressiveness

A *decision problem* is a set of pairs of the kind $\langle s, A \rangle$, where s is a problem instance and A is equal to either $True$ or $False$, and is the answer for the instance.

The class P is defined as the set of decision problems solved by deterministic polynomial-time bounded Turing machines. The class NP is defined as the set of decision problems solved by non-deterministic polynomial-time bounded Turing machines.

If X and Y are decision problems, a polynomial many-one reduction from X to Y is a function f computable by a deterministic polynomial-time bounded Turing machine such that $\langle s, True \rangle \in X$ if and only if $\langle f(s), True \rangle \in Y$.

A problem X is NP-hard if for all problems $Y \in$ NP there exists a polynomial many-one reduction from Y to X. If, moreover, $X \in$ NP, then X is said to be NP-complete.

Queries are defined ([13, 14]) as transformations defined on relational databases: Let U be some "universal" domain of constants. A relational database is a structure DB of the form $(D, R_1, \ldots, R_k)$ where $D \subset U$ is a finite set of domain constants and R_i is a relation of arity a_i over D for some integer a_i (i.e., $R_i \subseteq D^{a_i}$). DB is said to be of type $\bar{a} = (a_1, \ldots, a_k)$. The set D is called the *active domain* of the database.

A *computable query of type* $\bar{a} \rightarrow$ *bool* is a mapping

$$\mathcal{Q} : \{DB \mid DB \text{ is of type } \bar{a}\} \rightarrow \{True, False\}$$

satisfying the following constraints:

1. $\mathcal{Q}$ is partial recursive;
2. $\mathcal{Q}$ is generic, i.e., for each bijection ρ over D, $\rho(\mathcal{Q}(DB)) = \mathcal{Q}(\rho(DB))$.

Having defined computable queries, we draw our attention to their complexity. Let $\mathcal{Q}$ be a query of type $\bar{a} \rightarrow Bool$. Then $\mathcal{Q}$ is in NP if deciding whether $\mathcal{Q}(DB) = True$ is in NP, for any input database DB of type $\bar{a}$.

The *expressive power* of a formalism for querying relational databases is given by the set of queries it defines. Interesting classes of queries for classifying the expressive power of query languages are those defined by complexity classes, e.g., NP. We say that a language $\mathcal{L}$ *captures* NP if the set of queries defined by expressions from $\mathcal{L}$ coincides with the set of all queries in NP.

2.2 Datalog^{CIRC}

We illustrate the syntax and the semantics of DATALOG^{CIRC}, and briefly address its computational properties. Such notions will be used to introduce formal properties of NP-SPEC in Section 3.

2.2.1 Syntax of DATALOG^{CIRC}

First of all, we remind the syntax of DATALOG. We assume the existence of countable and distinct sets of constants, variables and predicate names. A DATALOG program [15] is a finite set T of universally quantified function-free first-order definite rules, i.e., sentences of the form:

$$a(\mathbf{t}) \quad \leftarrow \quad b_1(\mathbf{t_1}), \ldots, b_n(\mathbf{t_n})$$

where $n \geq 0$, $a(\mathbf{t})$ and $b_i(\mathbf{t}_i)$ $(1 \leq i \leq n)$ are atoms. An *atom* is formed out of a predicate name with arity k, for some $k \geq 0$, and a list of k arguments, which may include variables and constants. An atom is *ground* if no variables occur in it. A ground DATALOG rule such that $n = 0$ is called a *fact*. The set of clauses occurring in T is then naturally partitioned into a subset D of facts which constitute the *extensional component*, or *database*, and a subset π of non-ground or non-atomic formulae, called the *intensional component* of T. We assume that no constants appear in π, since constants in π can be represented by introducing ad-hoc unary predicates, and for each such predicate a single fact in the database. The set of constants appearing in the extensional component of a DATALOG program T is called the *active domain* of T. Let $r \in T$ be a DATALOG rule. The set of its *ground instance rules* is the set of ground rules obtained from r by consistently substituting its variables with constants belonging to the active domain of T.

A DATALOG^{CIRC} program is a triplet $\langle T; P; Q\rangle$ where T is a DATALOG program and $(P; Q)$ is a partition of the predicates occurring in T. Predicates in P are called *minimized*; predicates in Q are called *guessed*. The predicates in Q can appear only in the body of the clauses; furthermore, for each predicate p which belongs to P, there must be at least one clause having p in the head which determines its extension, unless p is a database predicate that denotes a database input relation through a set of facts.

With DATALOG^{CIRC} programs we can specify boolean queries, which are posed by means of *restricted clauses*, which are first-order formulae of the kind:

$$((\forall \mathbf{X}) A_1(\mathbf{X}) \wedge \cdots \wedge A_n(\mathbf{X})) \rightarrow \mathit{fail}$$

where each $A_i(\mathbf{X})$ $(1 \leq 1 \leq n)$ is an atom, $\mathbf{X}$ is a list of variables, and *fail* is a 0-ary predicate symbol, i.e., a propositional letter.

2.2.2 Semantics of DATALOG^{CIRC}

We start again with the semantics of DATALOG. The *Herbrand base* of T is the set of all ground atomic formulae constructed from predicate symbols and the

active domain of T. An *Herbrand interpretation* of a DATALOG program T is a subset of its Herbrand base. Let A be a ground atom and M be an Herbrand interpretation of a DATALOG program T. Then we say that A is true in M, written $M \models A$, if $A \in M$. If $M \not\models A$, then we say that A is false in M. The negative literal $\neg A$ is true in M if $M \not\models A$. An Herbrand interpretation M of a DATALOG program T is a *model* for T if for each ground rule C obtained by instantiating a rule in T with constants of the active domain, either the head of C is true in M or at least one literal in the body of C is false in M. The semantics of restricted clauses can be defined in the same way.

As for DATALOG^{CIRC} programs, their semantics originates from the non-monotonic formalism of *circumscription* [16], and takes into account partition of predicates into $(P;Q)$. Intuitively, the predicates in Q are those that are "guessed" by the interpretation, whereas those in P are either calculated or coming from the input database. In fact, the extension of the predicates in the set Q is not determined *a priori*; that is, let $q \in Q$ be a guessed predicate of arity k; we consider as possible extension associated to q all the subsets of the Cartesian product $D^k = D \times \cdots \times D$ (k times), where D denotes the active domain of T. We call *instance of* Q one possible guessing of extensions for each predicate in Q. Note that also the predicates belonging to the set P that depend through some clauses upon predicates in Q may have, in the sense outlined above, multiple extensions.

More formally, let T be a DATALOG^{CIRC} program. A preorder among the Herbrand models of T is defined as follows.

Definition 2.1 ($(P;Q)$-minimal models, [17]) *Let M, N be two Herbrand models of a formula T. We write $M \leq_{(P;Q)} N$ if:*

1. *predicates in Q have the same extension in N and M;*
2. *for each predicate $p \in P$, the extension of p in M is a subset —possibly not proper— of the extension of p in N.*

An Herbrand model M is called $(P;Q)$-minimal *for T if there is no Herbrand model N of T such that $N \leq_{(P;Q)} M$ and $M \not\leq_{(P;Q)} N$.* •

The semantics of a DATALOG^{CIRC} program relies on its Herbrand $(P;Q)$-minimal models: Let T be a fact-free DATALOG^{CIRC} program, D an extensional database, and γ a restricted clause, we write $T \wedge D \models_{(P;Q)} \gamma$ if γ is true in all Herbrand $(P;Q)$-minimal models of $T \wedge D$.

It can be easily seen that DATALOG^{CIRC} is a generalization of DATALOG: A DATALOG program T is just a DATALOG^{CIRC} program $\langle T;P;\emptyset\rangle$ in which the set Q of guessed predicates is empty.

In DATALOG^{CIRC}, we can specify the "graph coloring" problem, by means of the fact-free program $\langle T_{COL};P;Q\rangle$ consisting in the following clauses:

$$fail \leftarrow edge(X,Y), color(C), coloring(X,C), coloring(Y,C) \quad (1)$$

$$colored(X) \leftarrow color(C), coloring(X,C) \quad (2)$$

where $P = \{colored, fail, color, edge\}$, $Q = \{coloring\}$. The extensional database D is constituted by ground atomic instances of the following predicates:

- *edge*, which is a symmetric predicate encoding the set of edges of the input graph in the obvious way;
- *color*, which is a predicate encoding the set of available colors, e.g., in 3-coloring *color* has three facts. For simplicity, we assume that colors are denoted using constant symbols occurring in instances of the *edge* relation.

The query γ is defined by the following restricted clause:

$$((\forall X)colored(X)) \quad \rightarrow \quad fail \tag{3}$$

The intended meaning of the antecedent of the query γ is that we are interested in $(P;Q)$-minimal models of $T_{COL} \wedge D$ which assign a color to each element of the Herbrand universe, i.e., to each node. Note that we don't care if a node is assigned more than one color: If no conflict arises with such an overloading, then we can select in an arbitrary way a color for a node among those which the node is associated with.

It can be formally shown that $T_{COL}, D \not\models_{(P;Q)} \gamma$ if and only if the input graph G is colorable using the available colors. In fact, $T_{COL}, D \not\models_{(P;Q)} \gamma$ iff there exists at least one $(P;Q)$-minimal model M of T_{COL}, D such that $M \not\models \gamma$. The latter condition holds iff, in M, the extension associated to the guessed predicate *coloring* covers the set of graph's nodes in a way such that *fail* is not implied. Finally, such condition holds iff the input graph is colorable using colors stored in the database relation *color*.

The semantics of DATALOG^{CIRC} can be immediately and naturally extended to *search* queries, i.e., queries whose result is a set of relations, rather than a simple boolean answer. This is done by considering a set of predicate symbols $R \subseteq Q$, called *carriers*. A *computed result* associated to $\langle T; P; Q\rangle$, R, and γ over D is:

$$\{c(\mathbf{t}) \mid c \in R,\ c(\mathbf{t}) \in M, M \not\models \gamma \text{ and } M \text{ is a } (P;Q)\text{-minimal model of } T \wedge D\}.$$

For simplicity, in this paper we always assume $R = Q$. For instance, considering the graph coloring example shown above, a computed result is a (non-deterministically chosen) correct coloring of the input graph, if any.

2.2.3 Computational properties of DATALOG^{CIRC}

As it is evident from the previous example, the problem of determining whether $T \wedge D \models_{(P;Q)} \gamma$ or not, where the input is D, is NP-hard. In [10] it is proven that:

- The *data complexity* of DATALOG^{CIRC}, i.e., the complexity of query answering measured in the size of the input extensional database only, is NP-complete.

- The *expressiveness* of DATALOG^{CIRC} is such that the language captures NP. This is proven showing that, for each problem A in NP, there are a *fixed* fact-free DATALOG^{CIRC} program $\langle T; P; Q \rangle$ and a *fixed* restricted clause γ such that for each instance d of A encoded as an input database D, it holds that $T \wedge D \not\models_{(P;Q)} \gamma$ iff d is a "yes" instance of A.

This means that DATALOG^{CIRC} is capable of specifying exactly all problems belonging to NP. We remind that DATALOG is capable of expressing 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 whose question is: "Is the cardinality of C even?").

3 Basic NP-SPEC

In this section we introduce the basic language fragment of NP-SPEC, define its semantics in terms of a translation into a DATALOG^{CIRC} specification, and prove that it has the same expressive power of the latter language. NP-SPEC extensions are discussed in the next section.

3.1 Basic NP-SPEC syntax

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. Intervals of integers (cf. the 3-coloring example of Section 1) are not allowed.

- 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:

```
Subset(<domain>, <predicate_id>).
```

  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 must have the same arity as `<predicate_id>`.

- 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. Stratification introduces a restricted form of negation by allowing negative literals in the body of rules [18]. 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 restricted clause necessary for specifying a problem, i.e., one or more rules with `fail` in the head.

As an example, we can specify the "3-coloring" problem in basic NP-SPEC in the following way.

```
// 3-coloring
DATABASE
   color = {red, green, blue};
   node  = {1,2,3,4,5,6};
   edge  = {(1,2), (1,3), (2,3), (6,2), (6,5), (5,4), (3,5)};
SPECIFICATION
   Subset(node >< color, coloring).                      //  1
   fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).     //  2
   fail <-- coloring(X,C1), coloring(X,C2), C1 <> C2.    //  3
   fail <-- node(X), NOT colored(X).                     //  4
   colored(X) <-- coloring(X,_).                         //  5
```

The following comments are in order:

- The input graph and the names for the available colors are defined in the `DATABASE` section.

- In the search space declaration (`1`) the user declares the predicate symbol `coloring` to be a guessed one of arity 2. All other predicate symbols are, by default, not guessed.

- The symbol "><" denotes the Cartesian product, "<>" is the *not equal* predicate, and "_" is a mute variable. Comments can be inserted using the symbol "//".

- Rule `2` corresponds to rule (1) of the coloring DATALOG^{CIRC} program of Section 2.2.2.

- Rule `3` states that a node is not allowed to have more than one color. It has no counterpart in the DATALOG^{CIRC} version of Section 2.2.2, because it has only the practical use of choosing one color for each node. The specification is correct even if we drop it (in this case, a solution can then be found by choosing arbitrarily, for each node, any of its colors).

- Rule 4 corresponds to the restricted clause (3) of the coloring DATALOG^{CIRC} program of Section 2.2.2.

- Rule 5 corresponds to rule (2) of the coloring DATALOG^{CIRC} program of Section 2.2.2.

Running this program on the NP-SPEC compiler produces the following output:

```
coloring: (1,blue), (2,green), (3,red), (4,blue),
          (5,green), (6,blue)
```

Summarizing, the main differences of NP-SPEC with respect to DATALOG^{CIRC} are the following:

- guessed predicates have a finite domain explicitly associated to them;

- negation of atoms is allowed in the body of the rules, as long as the program is stratified;

- the restricted clause used as a query is listed among the other rules;

- the six relational operators ('==', '<>', '>', '<', '>=', and '<=') are allowed.

These differences are motivated by the aim of improving the readability of the specifications. The domain of guessed predicates is introduced also for efficiency considerations. In fact, searching on the specified domain results to be generally much faster than exploring the full active domain. Additional second-order predicates, illustrated in Section 4, help in further reducing the size of the search space.

3.2 Basic NP-SPEC formal semantics

In this section we show how any NP-SPEC specification S = (DB, SP) can be naturally translated into a DATALOG^{CIRC} program $\langle T; P; Q\rangle$ and a restricted clause γ, thus giving its formal semantics.

First of all, we define the vocabulary $\mathcal{L}_\mathcal{T}$ of T.

- $\mathcal{L}_\mathcal{T}$ contains every constant in S.

- For each predicate symbol s in S, $\mathcal{L}_\mathcal{T}$ contains the predicate symbol s with the same arity as s. If s is a guessed predicate, then s belongs to Q, otherwise it belongs to P.

- For each guessed predicate g in S, the predicate out_g belongs to $\mathcal{L}_\mathcal{T}$ and is in P. Each predicate out_g is used to exclude those extension of g which are not subsets of the corresponding domain (cf. forthcoming rule (5) and set of tuples (4)).

- For each predicate s which occurs in at least one negative literal in the body of a rule of SP, $\mathcal{L}_\mathcal{T}$ contains two predicates with the same arity as s:

- def_s, belonging to P,
- co_s, belonging to Q.

Each couple of predicates def_s and co_s is introduced in order to simulate negation on the corresponding predicate s (cf. forthcoming rules (6-8)).

- $\mathcal{L}_\mathcal{T}$ contains six binary predicates in P, one for each relational operator.

We now define D, i.e., the database part of T. In the following, H denotes the set of input constants of S, i.e., its active domain, and $\mathtt{H}^n$ denotes its n-wise Cartesian product.

- D contains every tuple in DB.
- For each guessed predicate g of SP, let `Subset(e,g)` be the corresponding declaration, n the arity of g, and e the relation corresponding to the domain e. D contains the set of tuples:

$$\{out_g(\mathbf{t}) \mid \mathbf{t} \in \mathtt{H}^n \setminus e\} \tag{4}$$

- For each relational operator r, D contains the standard extension of r upon $\mathtt{H}^2$.

Note that the size of D is polynomial in the size of DB. We now turn to the rules of T.

- Each NP-SPEC rule is translated into an equivalent DATALOG^{CIRC} rule in which each occurrence of p is replaced with p and each occurrence of NOT p is replaced with co_p. Moreover, mute variables are substituted with fresh variables.
- For each guessed predicate g of SP there is a rule

$$fail \leftarrow g(\mathbf{X}), out_g(\mathbf{X}) \tag{5}$$

in T, where $\mathbf{X}$ is a list of variables of the appropriate length.

- For each predicate s occurring, in a negative literal, in the body of a rule of SP, there are the following three rules:

$$def_s(\mathbf{X}) \leftarrow s(\mathbf{X}) \tag{6}$$
$$def_s(\mathbf{X}) \leftarrow co_s(\mathbf{X}) \tag{7}$$
$$fail \leftarrow s(\mathbf{X}), co_s(\mathbf{X}) \tag{8}$$

where $\mathbf{X}$ is a list of variables of the appropriate length.

Finally, we define the restricted clause γ as:

$$((\forall \mathbf{X}) def_{p_1}(\mathbf{X}) \wedge \cdots \wedge def_{p_n}(\mathbf{X})) \rightarrow fail$$

where $\{def_{p_i}\}_{i=1,\ldots,n}$ is the set of all the predicates of the sort def_p occurring in T. This clause, and rules (6-8), enforce the extension of each couple of predicates s and co_s to be a partition of $\mathtt{H}^m$, where m is the arity of s and co_s.

In conclusion, a *computed result* of the basic NP-SPEC program $\mathtt{S}$ is $\{c(\mathbf{t}) \mid c \in Q,\ c(\mathbf{t}) \in M\}$, where M is a $(P;Q)$-minimal model of the DATALOG^{CIRC} program T built as described, such that $M \not\models_{(P;Q)} \gamma$.

In order to clarify how the above translation works, we show it on the `3-coloring` specification shown in Subsection 3.1. The basic NP-SPEC specification is translated into a DATALOG^{CIRC} program $\langle T_{3col}, P_{3col}, Q_{3col}\rangle$ and a restricted clause γ_{3col} in the following way:

- The vocabulary $\mathcal{L}_{T_{3col}}$ contains the set of constants:

  $$H = \{red, green, blue, 1, 2, 3, 4, 5, 6\}.$$

  As for the set of predicates, $\mathcal{L}_{T_{3col}}$ contains:

  - $color, node, edge, colored, fail, coloring$, with the same arity as the corresponding ones in `3-coloring`. The last one belongs to Q_{3col}, the remaining ones belong to P_{3col}.
  - $out_{coloring}$ (in P_{3col}, with arity two), since $coloring \in Q_{3col}$.
  - $def_{colored}$ (in P_{3col}, with arity one) and $co_{colored}$ (in Q_{3col}, with arity one), because of the negative occurrence of $colored$ in rule 4 of `3-coloring`.
  - The six binary predicates *eq*, *neq*, *g*, *l*, *ge*, and *le* (in P_{3col}), corresponding to the six relational operators.

- The database part of T_{3col} contains:

  - The `DATABASE` section of `3-coloring`.
  - The set of tuples:

    $$\{out_{coloring}(X,Y) \mid \langle X,Y\rangle \in H^2 \setminus \{1,2,3,4,5,6\} \times \{red, green, blue\}\}.$$

  - The standard extension for the predicates *eq*, *neq*, *g*, *l*, *ge*, and *le*.

- The set of rules of T_{3col} contains:

  - The translation of the `SPECIFICATION` section of `3-coloring`:

    $$\begin{array}{rcl} fail & \leftarrow & edge(X,Y), coloring(X,C), coloring(Y,C) \\ fail & \leftarrow & coloring(X,C_1), coloring(X,C_2), neq(C_1,C_2) \\ fail & \leftarrow & node(X), co_{colored}(X) \\ colored(X) & \leftarrow & coloring(X,A) \end{array}$$

– The rule:

$$fail \leftarrow coloring(X,C), out_{coloring}(X,C), \tag{9}$$

because *coloring* is a guessed predicate.

– The rules:

$$\begin{aligned} def_{colored} &\leftarrow colored(X) && (10)\\ def_{colored} &\leftarrow co_{colored}(X) && (11)\\ fail &\leftarrow colored(X), co_{colored}(X), && (12) \end{aligned}$$

because of the occurrence of `NOT colored(X)` in rule 4 of `3-coloring`.

- The restricted clause γ_{3col} is:

$$((\forall X)\, def_{colored}(X)) \rightarrow fail$$

A computed answer for `3-coloring` is $\{coloring(x,c) \mid coloring(x,c) \in M\}$, where M is a $(P_{3col}; Q_{3col})$-minimal model for T_{3col}, such that $M \not\models_{(P_{3col};Q_{3col})} \gamma_{3col}$.

Note that the absence of domains and of negation in DATALOG^{CIRC} introduces some differences between the DATALOG^{CIRC} version of graph coloring given in Subsection 2.2.2, and $\langle T_{3col}, P_{3col}, Q_{3col}\rangle$. The latter program contains two kinds of additional rules: rule (9), which constrains the extension of *coloring* to be a subset of $\{1,2,3,4,5,6\} \times \{red, green, blue\}$, and rules (10-12), which simulate negation of the predicate *colored.* The database part of T_{3col} is suitably enriched, also.

3.3 Basic NP-SPEC expressive power

The following theorem states that basic NP-SPEC allows to specify the same set of problems as DATALOG^{CIRC}.

Theorem 3.1 DATALOG^{CIRC} *and basic* NP-SPEC *have the same expressive power.*

Proof. Basic NP-SPEC semantics is defined in terms of a corresponding DATALOG^{CIRC} program, so it is clear that each basic NP-SPEC program can be translated into a DATALOG^{CIRC} program. The translation of Section 3.2 is not modular, in the sense that the database of the DATALOG^{CIRC} program is a function of both the database and the specification of the NP-SPEC specification. Nevertheless, it can be easily seen that the size of the resulting database is polynomial in the size of the NP-SPEC database.

As for the other direction, we will show that each DATALOG^{CIRC} specification Σ, i.e., a program $\langle T; P; Q\rangle$ where $T = D \cup \pi$, and a restricted clause γ, can be expressed as a specification in basic NP-SPEC as well. The proof is in two steps:

1. We build a basic NP-SPEC program $\mathtt{P} = (\mathtt{DB}, \mathtt{SP})$, that we call the *reduction* of Σ.

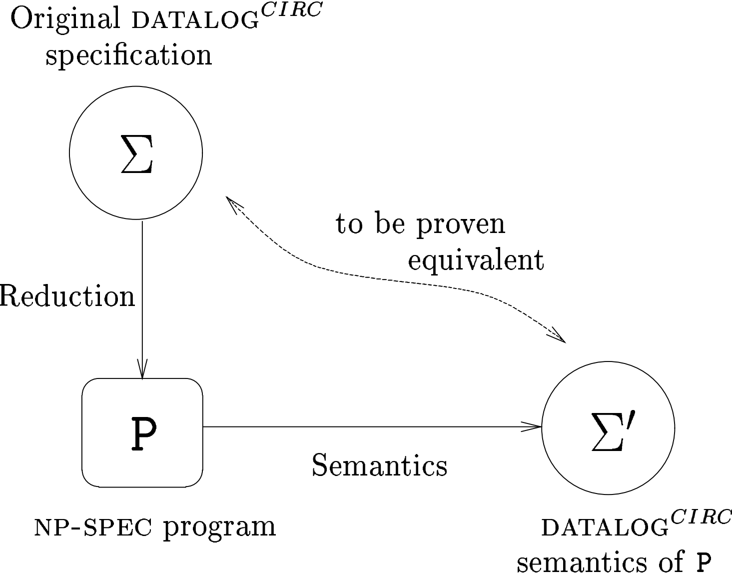

Figure 1: Proof sketch

2. Let $\Sigma' = \langle T'; P'; Q' \rangle, \gamma'$ be the DATALOG^{CIRC} specification representing the semantics of `P`. In order to prove the theorem we have to show that Σ and Σ' are equivalent (see Figure 1).

Step 1. We define the NP-SPEC database `DB` and an NP-SPEC specification `SP` as follows:

- Let H be the active domain of T. `DB` is $D \cup$ `DOM`, where `DOM` is a unary relation including all elements in H.
- For each predicate $q \in Q$, the clause `Subset(DOM >< ··· >< DOM,q)` is in `SP`, where the number of factors in the domain of `q` equals the arity of q.
- Let γ be:
  $$((\forall \mathbf{X}) A_1(\mathbf{X}) \wedge \cdots \wedge A_n(\mathbf{X})) \rightarrow e$$
  where $\mathbf{X}$ is $x_1, \ldots, x_k$.

  The following clauses are in `SP`:

  - `fail <-- e.`
  - `fail <-- NOT Ai(x1,...,xk).`, for each i $(1 \leq i \leq n)$.
- All rules in T are translated into identical `SP` rules.

Step 2. According to the semantical rules listed in the previous subsection, $\Sigma' = \langle T'; P'; Q' \rangle$ and γ' are defined as follows. Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be the

set of predicates occurring in γ. According to the transformations defined before,the set of predicates of T' contains all the predicates of T, plus predicates $\{co_a, def_a \mid a \in \mathcal{A}\} \cup \{out_g \mid g \in Q\} \cup \{fail\}$. As for the partition of predicates, P' is $P \cup \{def_a \mid a \in \mathcal{A}\} \cup \{fail\}$, and Q' is $Q \cup \{co_a \mid a \in \mathcal{A}\}$.

T' contains the following rules:

- All the rules of T.
- For each $a \in \mathcal{A}$, the rules:

$$\begin{aligned} def_a(\mathbf{X}) &\leftarrow a(\mathbf{X}) & (13)\\ def_a(\mathbf{X}) &\leftarrow co_a(\mathbf{X}) & (14)\\ fail &\leftarrow a(\mathbf{X}), co_a(\mathbf{X}) & (15)\\ fail &\leftarrow co_a(\mathbf{X}) & (16)\end{aligned}$$

  Rules (13-15) must be specified because a occurs in a negative literal (cf. rules (6-8)). Rule (16) is the translation of the rule `fail <-- NOT A(x1,...,xk)`. Note that rule (15) is actually redundant.

- For each $g \in Q$, the rules:

$$fail \leftarrow g(\mathbf{X}), out_g(\mathbf{X}) \quad (17)$$

- The rule:

$$fail \leftarrow e \quad (18)$$

Moreover, γ' is:

$$((\forall \mathbf{X}) def_{a_1}(\mathbf{X}) \wedge \cdots \wedge def_{a_n}(\mathbf{X})) \rightarrow fail$$

Finally, D and D' are the same, since predicates of the sort out_g have empty extension.

In this step we have to prove that:

a) for each $(P;Q)$-minimal model M of Σ such that $M \not\models_{(P;Q)} \gamma$ there exists a $(P';Q')$-minimal model M' of Σ' such that $M' \not\models_{(P';Q')} \gamma'$ and $\{c(\mathbf{t}) | c \in Q, c(\mathbf{t}) \in M\} = \{c(\mathbf{t}) | c \in Q, c(\mathbf{t}) \in M'\}$,

b) for each $(P';Q')$-minimal model M' of Σ' such that $M' \not\models_{(P';Q')} \gamma'$ there exists a $(P;Q)$-minimal model M of Σ such that $M \not\models_{(P;Q)} \gamma$ and $\{c(\mathbf{t}) | c \in Q, c(\mathbf{t}) \in M\} = \{c(\mathbf{t}) | c \in Q, c(\mathbf{t}) \in M'\}$.

Proof of point a). According to our hypothesis, we have a $(P;Q)$-minimal model M of Σ s.t. $M \not\models \gamma$. We must prove that we can build a $(P';Q')$-minimal model M' of Σ' (with the same extension for Q predicates as M) s.t. $M' \not\models_{(P';Q')} \gamma'$.

For each predicate p, let $M(p)$ denote the extension of a in M. The extension of co_a, (for each $a \in \mathcal{A}$) in M' is the relation $(H^k \setminus M(a))$ (where k is the arity of a), and the extension of def_a is H^k. The extension of the other predicates is the same as in M. We now prove that M' is a model for Σ'. Note that *fail* and e are not true in M', therefore $M' \not\models \gamma'$. Moreover, M' satisfies the rule $fail \leftarrow e$ and all the rules of the type (13-17). Rules of T are clearly satisfied by M' because they are satisfied by M. Therefore, $M' \models T'$.

We now prove that M' is $(P'; Q')$-minimal for Σ'. Since M and M' coincide w.r.t. predicates of $T \cup D$, $P \subseteq P'$, and the rules of T are a subset of the rules of T', the only way to decrease the extension of some predicate of P', satisfying all the rules of T', is to eliminate some tuples from the extension of a predicate of the form def_a for at least one $a \in \mathcal{A}$. This cannot be done without decreasing the extension of either a or co_a, but co_a is in Q', and its extension can not be changed. The extension of a can not be decreased as well, because we are assuming that M is $(P;Q)$-minimal, and a belongs to P. Therefore, we must conclude that M' is a $(P'; Q')$ minimal model for T'. Obviously, the extension associated to M' of predicates belonging to Q is the same as in M.

Proof of point b). As for the other direction, suppose that we have a $(P'; Q')$ minimal model M' for Σ' s.t. $M' \not\models_{(P';Q')} \gamma'$. We can build a model M for Σ, simply projecting out from M' atoms with predicates symbols not belonging to T. Note that:

- M is a $(P;Q)$-minimal model for $T \cup D$;
- $M \not\models \gamma$: in fact, since $M' \not\models_{(P';Q')} \gamma'$, $fail \notin M'$; in order to satisfy the rule $fail \leftarrow e$, also $M' \not\models e$ holds. Moreover, the extension of each predicate a occurring in γ coincides with H^k (being k the arity of a): should this not be true, the extension of co_a in M' would be non-empty, and due to the rule (16) associated to a, $M' \models fail$ would hold, thus satisfying γ'.

Q.E.D.

4 Full NP-SPEC

This section is devoted to show some more advanced features of NP-SPEC. Such features, which are a suitable set of built-in metapredicates called *tailoring predicates* and a set of arithmetical and aggregation operators, have been designed with the following three goals in mind:

1. allowing the user to specify a problem in a more natural way;
2. allowing the user to specify the search space in a way so as to help the interpreter generate an efficient code;
3. not exceeding NP as the data complexity.

As for the last point, in principle we could obtain the analog of Theorem 3.1 for the full language, because a program in full NP-SPEC can be translated into

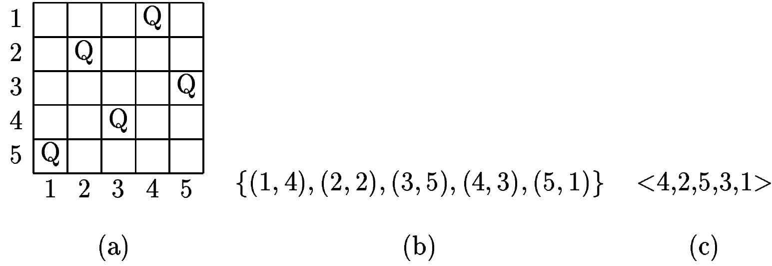

Figure 2: A solution of the 5-queens problem (a) and its representations as a subset of $\{1,\dots,5\} \times \{1,\dots,5\}$ (b) and as a permutation of $\{1,\dots,5\}$ (c)

a DATALOG^{CIRC} program of polynomial size. The only aspect we have to be careful about is the presence of intervals: since they are only syntactic sugar for the enumeration of their elements, a declaration like "`DOM = {1..1000};`" in the specification section must be expanded when measuring the size of the input.

4.1 Tailoring predicates

As we have seen in the previous sections, NP-SPEC provides a way to specify the structure of a search space by means of an appropriate declaration. As we shall explain in the forthcoming Section 6, the solution algorithm implemented by the prototype is basically a "guess-and-check" mechanism, that generates all possible elements of the search space, and checks whether they correspond to a solution.

In basic NP-SPEC, the only linguistic mechanism for declaring the structure of a search space is the `Subset` metapredicate, that, although in theory is sufficient to specify all problems in NP (cf. Theorem 3.1), it is in practice sometimes difficult to use. As an example, let us consider the famous n-queens problem, in which the goal is to place n non-attacking queens on a $n \times n$ chessboard (cf. Figure 2a).

In basic NP-SPEC, the specification of the problem is the following.

```
DATABASE
   NB_QUEENS = 5;
   row = {1..NB_QUEENS};
   column = {1..NB_QUEENS};
SPECIFICATION
   Subset(row >< column, queens).                // queens(R,C) <-> there is
                                                 // a queen in row R,column C
   good_row(R) <-- queens(R,_).
   good_column(C) <-- queens(_,C).
   fail <-- row(R), NOT good_row(R).             // >= 1 queens per row
   fail <-- queens(R,C1), queens(R,C2), C1 <> C2. // <= 1 queens per row
   fail <-- column(C), NOT good_column(C).       // >= 1 queens per column
   fail <-- queens(R1,C), queens(R2,C), R1 <> R2. // <= 1 queens per column
```

```
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
```

In the above specification, the search space is represented as a subset of the Cartesian product $\{1, \ldots, n\} \times \{1, \ldots, n\}$ (cf. Figure 2b). With such a search space, several rules, i.e., the first six, are needed to forbid attacks on the same row and on the same column. The last two rules check conflicts on diagonals.

In NP-SPEC it is possible to specify the search space as a *permutation* of the set $\{1, \ldots, n\}$ (cf. Figure 2c). The obvious advantage is that attacks on the same row and on the same column are impossible, and only conflicts on diagonals are to be checked. The specification section is now much more readable:

```
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
```

Another advantage concerns efficiency of the program: the latter version generates code faster than the former, because the size of the search space is smaller ($n!$ vs. 2^{n^2}).

In the current implementation, the following kinds of declarations can be put in the specification section:

- `Permutation(<domain>, <predicate_id>).`
- `Partition(<domain>, <predicate_id>, n).`
- `IntFunc(<domain>, <predicate_id>, min..max).`

where `<domain>` is a domain as specified in Section 3.1, and `<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 (cf. the graph coloring example in Section 1). 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.

Several examples of the usage of tailoring predicates are provided in Section 5.

4.2 Arithmetic operators

As we implicitly showed in the example concerning the n-queens problem, NP-SPEC allows for arithmetic operators (such as '+', '-', and '*') to be used in specifications.

Obviously, the operators are evaluated only for ground operands, belonging to the active domain.

NP-SPEC includes also SQL-style aggregation operators such as COUNT, SUM, MIN, and MAX. Such operators take two arguments, the first being an atom representing the predicate upon which the aggregation takes place, and the second one being the variable that stores the results. The character '*' identifies the argument to be aggregated.

For example, the atom SUM(p(*,_), N) means that N is the sum of the values of the first argument of the extension of p. The predicate involved in the aggregation can also be partially instantiated. For example, the following rule

```
h(Y) <--  COUNT(p(Y,*), N), N >= 4.
```

means that h(Y) is true if the number of instantiations of p with Y as first argument is greater than or equal to four. It is obviously impossible to use recursion and aggregates at the same time, e.g., having both predicate p in the head and the aggregate operator COUNT(p(...),...) in the body of the same rule.

Other examples of the usage of aggregation operators can be found in Section 5.4.

4.3 Full NP-SPEC formal semantics

The semantics of metapredicates can be easily given by translating them into basic NP-SPEC, as follows (d is a unary domain and p is a binary predicate symbol):

- IntFunc(d,p,min..max) is translated into a declaration and a set of rules, as follows:

```
Subset(d >< {min..max}, p).
p1(X) <-- p(X,_).
fail  <-- d(X), NOT p1(X).
fail  <-- p(X,Y1), p(X,Y2), Y1 <> Y2.
```

p1 is a fresh unary predicate symbol. As shown in the previous subsection, IntFunc generalizes Partition, hence the translation works for the latter metapredicate also.

- Permutation(d,p) is translated into a declaration and a rule, as follows (n is the cardinality of d):

```
Partition(d,p,n).
fail <-- p(X1,Y), p(X2,Y), X1 <> X2.
```

The case of non-unary domains can be easily obtained using the same technique.

Arithmetic operators are preinterpreted as database relations with the natural semantics. In order to remain in the NP class, they are defined only on the Herbrand Universe, or a polynomial-size extension of it. In fact usage of arithmetic can in principle lead to an increase in data complexity, thus exceeding NP. Anyway we note that NP remains the upper bound if the size of each arithmetical computation is polynomial in the size of the input database.

The translation of atoms containing aggregation operators into basic NP-SPEC needs a total order on the Herbrand universe, encoded as a database relation. In other words, the database must contain a declaration encoding the successor relation, such as:

```
successor  = {(c1,c2), ..., (cn1,cn)};
```

SUM(p(*),X), where p is a unary predicate symbol, is translated by:

1. substituting its occurrence with sum_p(X), and
2. adding the following rules:

```
aux_p(0,c1)  <-- NOT p(c1).                                    // 1
aux_p(c1,c1) <-- p(c1).                                        // 2
aux_p(X,Stage) <-- NOT p(Stage), successor(Stage1,Stage),
                   aux_p(X,Stage1).                            // 3
aux_p(X,Stage) <-- p(Stage), successor(Stage1,Stage),
                   aux_p(Y,Stage1), X == Y + Stage.            // 4
sum_p(X) <--  aux_p(X,cn).                                     // 5
```

The basic idea is to accumulate on the second argument of aux_p the sum of numbers o of the Herbrand universe such that p(o) holds. Rules 1 and 2 deal with the first element c1; rules 3 and 4 recursively deal with the successor of an element; rule 5 deals with the last element cn.

As for the translation of COUNT(p(*),X), its occurrence must be substituted with count_p(X), and three rules are to be modified as follows:

```
aux_p(1,c1) <-- p(c1).                                      // 2'
aux_p(X,Stage) <-- p(Stage), successor(Stage1,Stage),
                   aux_p(Y,Stage1), X == Y + 1.             // 4'
count_p(X) <--  aux_p(X,cn).                                // 5'
```

If a predicate `r` with arity $a > 1$ is involved in an aggregation operator, the above idea still works, as long as `r` has a key. As an example, let's consider a predicate `r` with arity 2, such that the first argument is the key, and the sum is made on the second argument, i.e., `SUM(r(_,*),X)`. The latter atom must be translated into `sum_r(X)`, and the translation consists in the following rules:

```
aux_r(0,c1) <-- NOT r(c1,_).                                  // 1''
aux_r(W,c1) <-- r(c1,W).                                      // 2''
aux_r(X,Stage) <-- NOT r(Stage,_), successor(Stage1,Stage),   // 3''
                   aux_r(X,Stage1).
aux_r(X,Stage) <-- r(Stage,W), successor(Stage1,Stage),       // 4''
                   aux_r(Y,Stage1), X == Y + W.
sum_r(X) <--  aux_r(X,cn).                                    // 5''
```

The translation of `MIN` and `MAX`, can be easily obtained with similar techniques.

Since the semantics of full NP-SPEC is given by a translation into basic NP-SPEC, its data complexity (hence its expressive power) does not exceed NP.

5 Specification examples

In this section we illustrate the specification in NP-SPEC of three classical NP-complete problems, namely *Integer knapsack*, *Subset sum*, and *Hamiltonian circuit* [9]. These examples show the use of various features of NP-SPEC and highlight the simplicity and compactness of specifications in NP-SPEC. We conclude the section showing a more complex example of a practical problem of university course timetabling [19].

Apart from those presented here, we have been able to specify in NP-SPEC a number of problems in NP in a simple way, the most significant of which are (in square parentheses we have reported the reference number in the Garey and Johnson's list [9])): *Inequivalence of simple functions* [PO15]; *Register sufficiency* [PO1]; *Dynamic storage allocation* [SR2]; *Pruned trie space minimization* [SR3]; *Integral flow with multipliers* [ND33]; *Consecutive ones matrix augmentation* [SR16]; *Boyce-Codd normal form violation* [SR29]; *Quadratic Diophantine equations* [AN8]; *Generalized instant insanity* [GP15]; *Modal logic S5 satisfiability* [LO13]; *Code generation for parallel assignments* [PO6]; *Balanced complete bipartite subgraph* [GT24]; *Intersection graph basis* [GT59]; *Traveling salesman problem* [ND22]; *Minimum broadcast time* [ND49].

5.1 Integer Knapsack

A *knapsack* of fixed capacity B, and a set of objects with their associated size and value are given. The *integer knapsack problem* consists in finding a selection

of the given objects so that the capacity of the knapsack is not exceeded and a given minimum total value K is reached.

Differently from the simple knapsack problem, i.e., *0-1 knapsack*, each object is available in unlimited quantity, therefore multiple copies of any object can be included in the solution.

The underlying NP-complete decision problem requires to decide whether such an assignment exists. Its mathematical specification is as follows, where $c(u)$ is the number of copies of the object u that are inserted in the knapsack.

INSTANCE:	Finite set U, for each $u \in U$, a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$, and positive integers B and K
QUESTION:	Is there an assignment of a non-negative integer $c(u)$ to each $u \in U$ such that $\Sigma_{u \in U} c(u)s(u) \leq B$ and such that $\Sigma_{u \in U} c(u)v(u) \geq K$?

In NP-SPEC we represent the input by means of the three arguments of the ternary relation `data` that encode the name of the object, its value, and its size, respectively. We write this specification in the following way:

```
SPECIFICATION
   IntFunc(u,take_up,0..B).
     // the maximum number of copies is always less than or equal to B
   table(Obj,Value,Size) <-- take_up(Obj,Num), data(Obj,V,S),
                             Value == Num * V, Size == Num * S.
   fail <-- SUM(table(_,_,*),S), S > B.
   fail <-- SUM(table(_,*,_),V), V < K.
```

where the first argument of the relation `take_up` represents the object and the second one is the number of copies in the assignment c.

A possible problem instance is described by the following `DATABASE` specification.

```
DATABASE
   u = {a,b,c,d,e};
   data = {(a,8,15), (b,5,7), (c,7,11), (d,3,4), (e,6,10)};
          // encoding is (u, v(u), s(u))
   B = 210;
   K = 135;
```

The most natural way to specify this problem turned out to be the tailoring predicate `IntFunc`, rather than `Partition` used in Section 1 for the 3-coloring problem. In the next two examples the most suitable ones are the predicates `Subset` and `Permutation`, respectively.

Notice that in the `DATABASE` section we can use also non-numeric constants such as strings.

5.2 Subset sum

Given a set of objects with an associated size, the subset sum problem consists in the selection of a subset with a given total size. Its formal specification is the following:

INSTANCE:	Finite set A, a size $s(a) \in Z^+$ for each $a \in A$, and a positive integer B.
QUESTION:	Is there a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?

The corresponding NP-SPEC specification, including an instance, is the following.

```
DATABASE
   size = {(a,1),(b,2),(d,4),(e,7),(f,5)};
   SUBSET_SIZE = 10;
SPECIFICATION
   Subset(size,taken).
   fail <-- SUM(taken(_,*),TotalSize), TotalSize <> SUBSET_SIZE.
```

5.3 Hamiltonian circuit

The Hamiltonian circuit problem consists in finding a circuit of edges that passes through all nodes of a given undirected graph. For the sake of brevity, we omit its formulation, that can be found in [9].

In the NP-SPEC specification for this problem, it is natural to use the tailoring predicate `Permutation`, which refers to the ordering of the nodes in the circuit. Specifically, the atom `order(V,M)` means that `V` is the `M`-th node in the circuit.

```
DATABASE
   N = 5;
   edge = {(1,2), (1,3), (2,4), (4,3), (4,1), (3,5), (5,1)};
SPECIFICATION
   Permutation({1..N},order).
   fail <-- order(X,P), order(Y,R), R == P+1, NOT edge(X,Y), NOT edge(Y,X).
   fail <-- order(X,N), order(Y,1), NOT edge(X,Y), NOT edge(Y,X).
```

The last two clauses simply check that there is an arc between a node and its successor in the path, and between the last and the first node. The predicate `edge` is used twice, with the arguments inverted, to make the relation *edge* symmetric (remind that the input graph is undirected).

5.4 University course timetabling

We now show how NP-SPEC can be used to specify the *university course timetabling* problem (see [19]). The course timetabling problem consists in the weekly

scheduling for all the lectures of a set of university courses in a given set of classrooms, avoiding the overlaps of lectures having common students.

We consider the basic decision problem (which is still NP-complete). Other variants involve more complex constraints and usually consider an objective function to be minimized (see [20]).

There are Q courses, P periods, and R rooms. For all $q \in 1, \ldots, Q$, the are k_q required lectures and each lecture $i \in 1, \ldots, k_q$ must be assigned to a period $p \in 1, \ldots, P$ and a room $r \in 1, \ldots, R$ in such a way that the following constraints are satisfied:

Conflicts: There is a conflict binary matrix C of size $Q \times Q$. If $c_{ij} = 1$ then courses i and j have common students, and cannot be scheduled at the same period.

Availabilities: There is an availability binary matrix A of size $Q \times P$. If $a_{ij} = 1$ then lectures of course i cannot be scheduled at period j.

Room Capacity: Each course $i \in 1, \ldots, Q$ has a number of students s_i and each room $j \in 1, \ldots, R$ has a capacity c_j. If $c_j < s_i$ no lecture of course i can take place in room j.

The problem is specified by assigning for each pair period/room (p, r) a value between 0 and Q, that represents the course that takes place in period p in room r. The value 0 represents the fact that r is unused at period p. We now provide the NP-SPEC specification, together with a small sample instance.

```
DATABASE
   NB_COURSES = 5;
   NB_ROOMS = 3;
   NB_PERIODS = 3;
        // course(c,n,s): course c requires n lectures (weekly)
        // and has s enrolled students
   course = {(1,1,80), (2,2,30), (3,1,40), (4,1,40), (5,2,40)};
        // conflict(c1,c2): c1 and c2 have common students
   conflict = {(1,2), (3,4)};
        // unavailable(c,p): teacher of course c is not
        // available for teaching at period p
   unavailable = {(1,1), (1,2)};
        // capacity(r,c):  room r has capacity c
   capacity = {(1,100), (2,50), (3,40)};

SPECIFICATION
        // timetable(p,r,c): in period p in room r there is a lecture
        // of course c. If timetable(p,r,0) is true, r is unused in p
   IntFunc({1..NB_PERIODS} >< {1..NB_ROOMS}, timetable, 0..NB_COURSES).

        // unavailability constraint
   fail <-- timetable(Period,_,Course), unavailable(Course,Period).
        // conflict constraint
```

```
fail <-- timetable(Period,_,Course1), timetable(Period,_,Course2),
         conflict(Course1,Course2).
     // capacity constraint
fail <-- timetable(_,Room,Course), capacity(Room,Size),
         course(Course,_,Students), Size < Students.
     // teaching requirements
fail <-- course(Course,M,_),
         COUNT(timetable(*,_,Course),N), N <> M.
     // no two lectures of the same course in the same period
fail <-- timetable(Period,Room1,Course), timetable(Period,Room2,Course),
         Course <> 0, Room1 <> Room2.
```

The resulting timetable is delivered by the extension of the predicate `timetable`, which stores for each pair period/room the lecture that takes place in that room in the given period.

6 The NP-SPEC compiler

NP-SPEC has been implemented in a system prototype, whose structure is shown in Figure 3. The prototype system is written in Gnu C. The NP-SPEC compiler was developed using FLex and Bison. In the present prototype release, both the specification and the database section of NP-SPEC specifications are written in text files.

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.

ECL^iPS^e [21] is a PROLOG compiler integrated with several extensions. The most notable ones are those which deal with constraint definition and propagation. In particular, we make use of the fd library, that deals with constraints of finite domain variables.

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

Some figures about the NP-SPEC prototype performances are given in the following Section 7.

7 Considerations on performance

The current version of the system is meant only for developing executable specifications, and not for effective program synthesis. Therefore, in this work we mostly focus on the language design and semantics rather than on the efficiency of the generated programs. Recently, some of the authors designed a new strategy, based on translating a specification into a SAT instance [22].

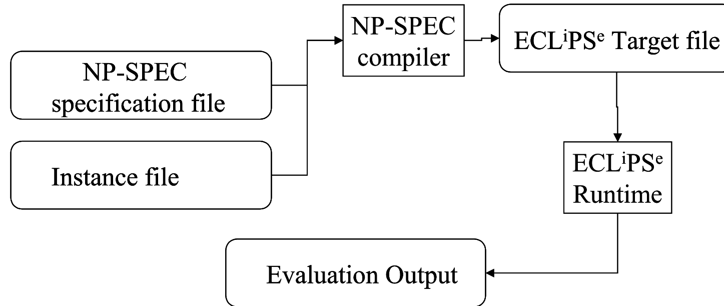

Figure 3: The NP-SPEC compiling & executing environment

Nevertheless, we believe that an evaluation of the efficiency of the system is still useful, and it prepares the ground for future improvements. For this reason, in this section we provide an experimental analysis of its performances, and we assess how it compares to state-of-the-art solvers and algorithms.

We tested the performance of our prototype by solving several problems on both randomly generated and non-random cases, as discussed in the next subsections. We performed our tests on a dual-processor Sparc Ultra 2 running at 200 MHz.

7.1 Evaluation on random XSAT instances

This test concerns randomly generated instances of the XSAT problem, i.e., the satisfiability problem for propositional formulae in CNF (aka SAT) when all clauses have exactly X literals. When X $\geq$ 3, the XSAT problem is NP-complete, cf. [9, problem LO1, page 259]. Algorithms are obviously written for SAT, but their performance is often tested on instances of XSAT, for various values of X, cf., e.g., the DIMACS competition [23]. It is known from the literature (cf., e.g., [24]) that, for many complete algorithms, the hardest instances for XSAT are close to the so-called "crossover point", i.e., the point in which the probability of a randomly generated formula to be satisfiable is about 50%. For such a reason, we generated instances close to the crossover point, and solved them using several programs. For each program, we found the size of the largest instance that can be solved in 1 minute. More precisely, for any given number of variables, we chose the number of clauses so as to be at the crossover point, and computed the average time on 100 instances, randomly generated according to the *fixed length* model described in [24]. We remind that the number of clauses at the crossover point depends primarily on X, and partially on the number of variables, cf. [25] (the number of clauses/number of variables ratio is about 4.3 for X=3, and about 370 for X=9).

Five different programs were compared:

1. the ECL^iPS^e program automatically generated by the NP-SPEC compiler;

X	NP-SPEC	C++ enumeration	ECL^iPS^e	`rel_sat_rand`	`satz`
3	17	24	110	255	320
9	10	17	10	24	N/A

Table 1: Largest instance of XSAT (number of variables) that can be solved in 1 minute, close to the crossover point

2. a hand written C++ program based on the same (enumeration) algorithm used by the above program;
3. a hand written ECL^iPS^e program which uses constraints in a more efficient way than the automatically generated one;
4. two state-of-the-art SAT solvers based on the Davis-Putnam [26] procedure and available from [27]:
   - `rel_sat_rand`, described in [28];
   - `satz`, described in [29].

Table 1 shows the results of the experiments. In instances with "short" clauses, i.e., X=3, the beneficial effects of the constraint propagation technique used by `rel_sat_rand`, `satz`, and the native ECL^iPS^e program are evident, and such solvers are several orders of magnitude faster than both the automatically generated program and the naïve C++ program.

On the other hand, when constraints are much more intricate, i.e., X=9, using sophisticated constraint propagation techniques becomes a burden rather than an advantage: `rel_sat_rand` and the native ECL^iPS^e program, have an efficiency comparable to enumeration algorithms. `satz` could not solve an instance with 10 variables and 1000 clauses, i.e., very far from the crossover point, in one day.

We conclude that:

- our prototype generates a quite inefficient program, when it is compared with programs implementing smart algorithms;
- the picture is rosier when our prototype is compared with:
  - hand-written programs implementing *exactly the same algorithm*, or
  - programs implementing smart algorithms, when they run on difficult instances.

As a final remark, we note that for under-constrained instances of 3SAT, i.e., instances with few clauses, finding a solution is always simpler, because typically a formula has many models. As an example, the NP-SPEC program was able to find a model of an instance with 50 variables and 60 clauses in about two hours.

7.2 N-queens and blocks world

We tested the performance of NP-SPEC on some other famous combinatorial problems which are often used as benchmarks for constraint solvers:

- the n-queens problem (cf. Section 4.1);
- the blocks-world problem, which is characterized by:
  - input:
    - n cubical blocks, one table large enough to hold all of them;
    - an initial configuration:
      - each block is either on another block or on the table;
      - each block is either clear or there is a single block on top of it;
    - a goal, i.e., the desired configuration of blocks;
  - a single kind of action: move a single clear block onto the table, or onto another clear block.

Our goal was to prove that the prototype was able to solve some small instances of both problems. Actually, NP-SPEC was able to solve, in few minutes, instances of the 12-queens problem, and instances of the blocks-world problem with 4 blocks.

As for comparison with other programs solving the same problems, conclusions are similar to those of the previous subsection, i.e., our programs performs poorly when compared to programs implementing smart algorithms, but performs similarly to hand-written programs implementing the same algorithm. As an example, a hand-written C++ program for n-queens implementing enumeration can solve in the same amount of time the instance with $n = 13$.

8 Related work

The language NP-SPEC is similar in spirit to the system KIDS [6], which produces the concrete implementation starting from the specification written in a logic language based on set theory. The main characteristic of KIDS is the use of a "domain theory" (written in terms of a set of axioms), which guides the system in the application of a predefined set of transformation rules that lead to the synthesis of a program compliant to the specified strategy.

Similar to our proposal, KIDS makes use of an abstract specification language "sensible" to the implementation issues. Moreover, in both languages there is the idea of selecting the search space which is the most suitable for the specific problem. In NP-SPEC, however, the concept of domain theory is absent, and this is due to the lack of a deductive mechanism. In fact, the translation of an NP-SPEC specification is done in a single step, and this is made possible by the relative limitedness of the expressiveness of NP-SPEC. Such limitation, which is characterized in an exact formal way (see Section 2.2.3), is one of the strengths

of NP-SPEC, together with the capability of a fully automatic translation (in KIDS a certain degree of interactivity is required).

Another specification language proposed in the literature is SPILL [7]. In few words, SPILL is a typed subset of pure PROLOG. The main limitation with respect to PROLOG is the so-called *groundness restriction*: at run time, terms must not contain variables and they must be finite. However, differently from NP-SPEC, SPILL allows the use of function symbols in the language. As another difference, its semantics is a pure first order one, as it does not include any sort of model minimization operations.

A specification in SPILL is also "executable", but not in the same sense of NP-SPEC. That is, in SPILL it is not possible to compute a solution of a problem from the specification, but rather only to *test* whether a provided solution is feasible, according to the specification. This limitation is a precise design decision, because SPILL, different from NP-SPEC, is not meant for computing, but to test the specification against some specific "interesting" cases. Finally, like KIDS, SPILL does not provide a characterization of its expressiveness and its complexity.

Another remarkable specification language is the one proposed by Minton [5] for the MULTI-TAC system. Such language is a sorted first order logic, which is specifically oriented to the specification of a problem by means of constraints. As for the other languages, no characterization is provided in the paper for the expressiveness of the language.

Finally, we note that there has been recently considerable interest in the implementation of rule-based deductive systems which semantics rely on non-monotonic formalisms (cf., e.g., [3, 4]). Apart from the semantic differences we already noticed in Section 1, we remark that in NP-SPEC the search space declaration is kept separate from the constraints of the problem, thus gaining in term of user-friendliness. In particular, the search space is declared by using special second-order predicates.

9 Conclusions and future work

We have presented NP-SPEC, an executable specification language for search problems, and we have shown how some classical problems can be expressed in NP-SPEC in a natural and concise way.

Differently from most specification languages in the literature, NP-SPEC has a precise characterization of its expressive power, namely the class NP. On the one hand, such expressiveness guarantees the decidability of the execution and, to a limited extent, its efficiency. On the other hand, it allows the designer of the system to implement optimized solution techniques.

As for the optimization of the performance, we saw in Section 7 that there are currently two sources of inefficiency in NP-SPEC:

- the algorithm adopts a blind guess-and-check strategy,
- the target language, i.e., ECL^iPS^e, is non-procedural.

The former source has much more impact on performances than the latter. As a consequence, we are currently investigating how to generate programs implementing smarter algorithms, e.g., backtracking-based execution strategies such as backjumping and forward-checking, which make use of partial solutions and/or heuristics. Two of the authors [22] have also proposed to translate a specification into an instance of the SAT problem, which can be solved by any state-of-the-art solver, available from the research community.

Finally, in the future we plan to improve the language extensions of NP-SPEC by including a limited number of additional metapredicates. The need for them might indeed arise from further experimentations with the language. This will be done with the twofold objective of allowing more natural specifications and improve the efficiency of their execution.

The home page of the NP-SPEC project is:

`http://www.dis.uniroma1.it/~cadoli/projects/NP-SPEC`

Acknowledgments

The first author has been in part supported by a scholarship from the Italian National Research Council (CNR) under the "Short-term mobility" program. The work has also been supported by ASI (Italian Space Agency) and MURST (Italian Ministry for University and Scientific and Technological Research) under the 40% "Interdata" project.

References

[1] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile, "NP-SPEC: An executable specification language for solving all problems in NP", in *Informal proceedings of the Eigth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'98)*, 1998.

[2] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile, "NP-SPEC: An executable specification language for solving all problems in NP", in *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*. 1999, number 1551 in Lecture Notes in Artificial Intelligence, pp. 16–30, Springer-Verlag.

[3] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, "The KR system `dlv`: Progress report, Comparisons and Benchmarks", in *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998, pp. 406–417.

[4] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm", *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3,4, pp. 241–273, 1999.

[5] Steven Minton, "Automatic configuring constraint satisfaction programs: A case study", *Constraints*, vol. 1, no. 1/2, pp. 7–43, 1996.

[6] Douglas R. Smith, "KIDS: A semi-automatic program development system", *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 1024–1043, 1990.

[7] Feliks Klużniak and Mirosława Miłkowska, "Spill – a logic language for writing testable requirements specifications", *Science of Computer Programming*, vol. 28, pp. 193–223, 1997.

[8] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Strategies for incorporating formal specifications in software development", *Communications of the ACM*, vol. 37, no. 10, pp. 74–86, 1994.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, Ca, 1979.

[10] Marco Cadoli and Luigi Palopoli, "Circumscribing DATALOG: expressive power and complexity", *Theoretical Computer Science*, vol. 193, pp. 215–244, 1998.

[11] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison Wesley Publ. Co., Reading, Massachussetts, 1995.

[12] C. H. Papadimitriou, *Computational Complexity*, Addison Wesley, Reading, MA, 1994.

[13] A. Chandra and D. Harel, "Computable queries for relational databases", *Journal of Computer and System Sciences*, vol. 21, pp. 156–178, 1980.

[14] A. Chandra and D. Harel, "Structure and complexity of relational queries", *Journal of Computer and System Sciences*, vol. 25, pp. 99–128, 1982.

[15] J. D. Ullman, *Principles of Database and Knowledge Base Systems*, vol. 1, Computer Science Press, 1988.

[16] J. McCarthy, "Circumscription - A form of non-monotonic reasoning", *Artificial Intelligence*, vol. 13, pp. 27–39, 1980.

[17] V. Lifschitz, "Computing circumscription", in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85)*, 1985, pp. 121–127.

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

[19] D. de Werra, "An introduction to timetabling", *European Journal of Operational Research*, vol. 19, pp. 151–162, 1985.

[20] Andrea Schaerf, "A survey of automated timetabling", *Artificial Intelligence Review*, vol. 13, no. 2, pp. 87–127, 1999.

[21] A. Aggoun and *et al*, *ECL^{i}PS^e User Manual (Version 4.0)*, IC-Parc, London (UK), July 1998.

[22] Marco Cadoli and Andrea Schaerf, "Compiling problem specifications into SAT", in *Proceedings of the European Symposium On Programming (ESOP 2001)*. 2001, vol. 2028 of *Lecture Notes in Computer Science*, pp. 387–401, Springer-Verlag.

[23] David S. Johnson and Michael A. Trick, Eds., *Cliques, Coloring, and Satisfiability. Second DIMACS Implementation Challenge*, vol. 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1996.

[24] B. Selman, D. Mitchell, and H. Levesque, "Generating Hard Satisfiability Problems", *Artificial Intelligence*, vol. 81, pp. 17–29, 1996.

[25] J. Crawford and L. Auton, "Experimental results on the crossover point in random 3-SAT", *Artificial Intelligence*, vol. 81, pp. 31–57, 1996.

[26] M. Davis and H. Putnam, "A computing procedure for quantification theory", *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[27] "SATLIB - The Satisfiability Library", http://www.informatik.tu-darmstadt.de/AI/SATLIB.

[28] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization", in *Proc. of the 15th Nat. Conf. on Artificial Intelligence (AAAI-98)*, 1998, pp. 431–437.

[29] C.M. Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems", in *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence (IJCAI-97)*, 1997, pp. 366–371.

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>
<letters>           ::= <letter> | <letters> , <letter>

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

<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> ) .
             | Partition ( <domain> , <predicate_id> ) .
             | Permutation ( <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
```

The syntax of basic NP-SPEC can be obtained just replacing the following rules:

```
<metafact>    ::=  Subset ( <domain> , <predicate_id> ) .
<literal>     ::=  <atom>
```