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

Marco Cadoli[1], Luigi Palopoli[2], Andrea Schaerf[1], Domenico Vasile[2]

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

2. Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
I-87036 Rende (CS), Italy
e-mail: {palopoli,vasile}@si.deis.unical.it

June 25, 1998

1 Introduction

In this paper we present the executable specification language NP-SPEC, which allows the user to specify exactly all problems which belong to the complexity class NP.

We start with 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., [GJ79]), and is formally defined as follows.

INSTANCE:	Graph $G = (N, E)$
QUESTION:	Is G 3-colorable, i.e., does there exist a function $col : N \rightarrow \{0, 1, 2\}$ 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 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 version of NP-SPEC, processing the above specification gives as output a PROLOG program. Running the PROLOG program produces the following output, which represents the colors to be assigned to the nodes so that the input graph is 3-colored.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

```
coloring: (1,0), (2,1), (3,2), (4,0), (5,1), (6,0)
```

The underlying language of NP-SPEC is a subset of PROLOG, and more precisely of its function-free fragment DATALOG. In particular:

- the clauses in the `DATABASE` section specify a set of *facts*, i.e., function-free ground atomic first-order formulas;
- a clause in the `SPECIFICATION` section can be a universally quantified function-free first-order definite formula.

The main difference between NP-SPEC and DATALOG is the possibility of using second-order logic in a limited way. Referring to the above example, the clause `Partition(NODE, coloring,3)` has, intuitively, the following meaning: `coloring` is a binary predicate, and its extension can be any set of the following kind:

$$\{\langle n, c\rangle \mid n \text{ is in the extension of } \texttt{NODE},\ c \in \{0,1,2\}^1\}.$$

The intuitive meaning of the `SPECIFICATION` section is: `fail` becomes true 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 (cf. atoms `coloring(X,C),coloring(Y,C)`).

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.

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 (with the appropriate arity) of the active domain. 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 [CP98].
- 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 as efficient as possible.

[1] This set has cardinality 3, as specified by the clause.

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 solution 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 discuss related work. More specification examples are reported in the appendix.

2 The Specification Language NP-SPEC

As we mentioned in the previous section, NP-SPEC is based on the language DATALOG^{CIRC}. Actually, the syntax of NP-SPEC extends the one of DATALOG^{CIRC} by means of the tailoring predicates, which are syntactic extension that makes it easier to specify a problem. For this reason, we start with the syntax and semantics of DATALOG^{CIRC}, and then explain the main tailoring predicates currently implemented. We also briefly address computational properties of DATALOG^{CIRC}, which explain why NP-SPEC is able to specify exactly all problems in NP.

2.1 Syntax of DATALOG^{CIRC}

First of all, we remind the syntax of DATALOG. A DATALOG program [Ull88] is a finite set of universally quantified function-free first-order definite formulae T, i.e., sentences of one of the forms reported below.

$$\begin{aligned} a(\mathbf{X}) &\leftarrow b_1(\mathbf{X_1}), \ldots, b_n(\mathbf{X_n}) \\ a(\mathbf{c}) &\leftarrow \end{aligned}$$

where $n \geq 0$, $a(\mathbf{X})$, $b_i(\mathbf{X_i})$ $(1 \leq i \leq n)$, and $a(\mathbf{c})$ are atoms. A clause of the latter kind is called a *fact*.

An atom is *ground* if no variables occur in it. The set of clauses occurring in T is then naturally partitioned in a subset D of facts which constitute the *input database*, and a subset π of non-ground or non-atomic formulae. We assume that no constants appear in π.

A DATALOG^{CIRC} program is a triple $\langle T; P; Q\rangle$ 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*. Intuitively the predicates in Q are those that are guessed by the algorithm, 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*, conversely it is necessary to consider for each of them all possible extensions; that is, all the subsets of the power-set of the Cartesian product of the domains of their attributes. In order to obtain all possible combination among the instances of all predicates of Q it is sufficient to consider their Cartesian product. We call *instance of* Q an element of the latter set. Note that also the predicates belonging to the set P that depend through some clauses upon predicates in Q may have multiple extensions.

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 the extension, unless p is a database predicate that represents a database input relation D through a set of facts.

The purpose of writing DATALOG^{CIRC} programs is to ask *boolean queries*. Queries are posed by means of *restricted clauses*, which are first-order formulae of the kind

$$(\exists \mathbf{X}) \neg A_1(\mathbf{X}) \vee \cdots \vee \neg A_n(\mathbf{X}) \vee e \tag{1}$$

where each A_i ($1 \leq 1 \leq n$) is a predicate symbol, and e is a 0-ary predicate symbol, i.e., a propositional letter. We note that formula (1) can be equivalently written as

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

2.2 Semantics of DATALOG^{CIRC}

First of all, let us recall the well-known semantics of DATALOG. A (Herbrand) model M of a DATALOG program T is a set of ground atomic formulae such that for each ground instance clause C constructible from T either the head of C is in M or at least one atom in the body of C does not belong to M. The intersection of all the Herbrand models of a DATALOG program T is itself a model for T [vEK76, Ull88]. This model, denoted LM_T, is called the *least* or *minimum* model of T. The semantics of a DATALOG program relies on its least model.

As for DATALOG^{CIRC} programs, their semantics originates from the formalism of *circumscription*, and takes into account partition of predicates into $(P;Q)$. 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, [Lif85]) *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 input 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} constitutes 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 specify the "3-coloring" problem, by means of the fact-free program $\langle T_{co3COL}; P; Q \rangle$ constituted by the following clauses:

$$\begin{array}{rcl}
\mathit{fail} & \leftarrow & \mathit{edge}(X,Y), \mathit{red}(X), \mathit{red}(Y). \\
\mathit{fail} & \leftarrow & \mathit{edge}(X,Y), \mathit{blue}(X), \mathit{blue}(Y). \\
\mathit{fail} & \leftarrow & \mathit{edge}(X,Y), \mathit{green}(X), \mathit{green}(Y). \\
\mathit{has_color}(X) & \leftarrow & \mathit{red}(X). \\
\mathit{has_color}(X) & \leftarrow & \mathit{blue}(X). \\
\mathit{has_color}(X) & \leftarrow & \mathit{green}(X).
\end{array}$$

where $P = \{\mathit{has\_color}, \mathit{fail}, \mathit{edge}\}$, $Q = \{\mathit{red}, \mathit{green}, \mathit{blue}\}$. The input database D is constituted by ground atomic instances of *edge*, which is a symmetric extensional predicate encoding the set E of edges of the input graph G in the obvious way. Consider next the query γ defined by the following restricted clause:

$$((\forall X) \mathit{has\_color}(X)) \quad \rightarrow \quad \mathit{fail}$$

The intended meaning of the last three rules of T_{co3COL} is to force each node to be colored in some way. The intended meaning of the antecedent of the query γ is that we are interested in $(P; Q)$-minimal models of $T_{co3COL} \wedge D$ which assign a color to each node. As a matter of fact, 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.

As it can be easily verified, $T_{co3COL}, D \not\models_{(P;Q)} \gamma$ if and only if the input graph G is 3-colorable.

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 [CP98] it is proven that:

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

This means that the language is capable of specifying exactly all problems which belong 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 which question is: "Is the cardinality of C even?").

2.4 Syntax of NP-SPEC: basics

In general, the syntactic differences between DATALOG^{CIRC} and NP-SPEC have been designed with 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.

The main differences between DATALOG^{CIRC} and NP-SPEC are the following:

- The restricted clause necessary for specifying a problem always consists in the single literal `fail`.
- It is possible to denote the complement of a predicate, by using the prefix "`co_`".
- There are suitable built-in metapredicates for specifying the search space (cf. Section 2.5).

In NP-SPEC, in order to identify the predicates belonging to Q and to fix the domains for their attributes, we include in the specification section, for each element of Q, a fact of the following form.

```
Subset(<DOMAIN>, <predicate_name>).
```

where Subset is a built-in metapredicate, and <DOMAIN> is a relation that identifies the domain upon which the extension of the predicate is guessed. <DOMAIN> is a finite set defined as an input relation, or as an enumeration, or by means of union, intersection, difference, and Cartesian product.

Using the basic syntax, we can specify the question of the "3-coloring" problem in the following way, which is close to the DATALOG^{CIRC} program of Section 2.2 (in this case, the user must specify the name of the colors).

```
SPECIFICATION
 Subset(NODE >< {red, green, blue}, coloring). // >< denotes Cartesian product
 fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).
 fail <-- coloring(X,C1), coloring(X,C2), C1 =/= C2. // =/= is the "not equal"
                                                    // predicate
 colored(X) <-- coloring(X,_).    // _ is a mute variable
 fail <-- node(X), co_colored(X). // co_colored is the complement of colored
```

The output of the program for the database of Section 1 is in this case the following.

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

The specification is correct even if we drop the third clause. In this case, a node is allowed to have more than one color, and a solution can then be found by choosing arbitrarily any of its colors for each node.

2.5 Extensions of NP-SPEC: Tailoring Predicates

It is clear that, using the specification language described above, the generated program must make use of a pure "guess-and-check" mechanism, that explores all instances of Q predicates.

We note that, depending on the specific problem under consideration, it might be helpful to exclude some of the instances of the predicates in Q because they are "structurally" infeasible. For example, it is possible for a given problem that the elements to be taken into account are all the permutations of a given set S, rather than the n-wise (n being equal to $|S|$) Cartesian product $S \times \cdots \times S$. This happens, as an example, for the N-queens problem, where the search space is a permutation of the set $\{1, \ldots, N\}$.

A search space based on permutations would in general result in a more efficient program than one based on a guess-and-check approach of the full Cartesian product. One way to represent permutations is to use specification clauses that invalidate inappropriate instances. For example, we may use a dyadic predicate Permute to represent all possible permutations of S: The first attribute represents a generic element of S and the second one the position it holds in the permutation. According to the semantics, all subsets of $S \times \{1, \ldots, |S|\}$ are possible extensions of Permute, but those that do not associate a position to an element, or associate more than one element to a position, do not correspond to a feasible element of the search space. Such elements could then be eliminated by using suitable clauses with Permute in the body and the atom fail in the head, such as the following one:

```
fail <-- Permute(X,I), Permute(Y,I), X =/= Y.
```

Such an approach, has the drawback of requiring the analysis of all $(P;Q)$-minimal models, which can outnumber the cardinality of the intended search space of the problem.

In order to overcome this drawback we introduce, in addition to Subset, more metapredicates, called *tailoring* predicates, that are used to specify that for a given predicate it is sufficient to consider only a certain class of extensions.

In the current implementation, the following metapredicates can be used:

- Partition(<DOMAIN>, <predicate_name>, n).
- Permutation(<DOMAIN>, <predicate_name>).
- IntFunc(<DOMAIN>, <predicate_name>, min..max).

where <DOMAIN> is a domain (cf. Section 2.4), and <predicate_name> is a predicate in Q.

The metapredicate Partition has a further integer-valued argument n that states the number of subsets in which the domain must be partitioned. The metapredicate IntFuct has a composite argument min..max that determine the codomain [min,...,max] of the function.

The predicate <predicate_name> should code the sets, or the collection of sets, built from <DOMAIN> according to the specific metapredicate used. Such coding is implemented using an extra internal domain, identified with an interval of integer values, called "ID domain" that has a different meaning for each metapredicate. The arity of <predicate_name> is augmented by one with respect to the arity of <DOMAIN>, in order to add an argument that ranges upon the ID domain.

As an example, if <DOMAIN> is $D = \{a, b, c, d\}$, the argument n for Partition is 3, and min and max are 1 and 30, respectively, the following are possible extension of the predicate used as argument of the metapredicate.

DOMAIN

IT
a
b
c
d

PART

IT	ID
a	1
b	1
c	0
d	2

PERM

IT	ID
a	2
b	0
c	1
d	3

INTF

IT	ID
a	27
b	5
c	15
d	9

The above instances represent, respectively, the partition of D into the three subsets $\{c\}$,$\{a, b\}$, $\{d\}$ of D, the permutation $[b, c, a, d]$ of D and the integer function f such that $f(a) = 27$, $f(b) = 5$, $f(c) = 15$, $f(d) = 9$.

Even with this small number of metapredicates we were able to specify in NP-SPEC a number of problems in NP in a natural way, the most significant of which are (in square parentheses we have reported the reference number in the Garey and Johnson's list [GJ79])): *Integer knapsack* [MP10]; *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* [GP24]; *Intersection graph basis* [GT59]; *Traveling salesman problem* [ND22]; *Minimum broadcast time* [ND49].

For the sake of brevity, we do not describe other features of the language NP-SPEC which improve its usability.

2.6 The NP-SPEC prototype

The NP-SPEC prototype is implemented mostly in SWI-PROLOG. The core of the system is composed of two modules. The first one, which is independent from the specification under consideration, implements the enumeration algorithm for the solution and takes care of the tailoring predicates and other secondary features of the language (e.g., the relational operators). The second module parses the specification and generates a PROLOG "translation" of it. The two modules are then compiled together automatically.

3 Related Work

The language NP-SPEC is similar in spirit to the system KIDS [Smi90], 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, it is present 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 a 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 2.3), 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 [KM97]. 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.

References

[CP98] Marco Cadoli and Luigi Palopoli. Circumscribing DATALOG: expressive power and complexity. *Theoretical Computer Science*, 193:215–244, 1998.

[GJ79] 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.

[KM97] Feliks Klużniak and Mirosława Miłkowska. Spill – a logic language for writing testable requirements specifications. *Science of Computer Programming*, 28:193–223, 1997.

[Lif85] V. Lifschitz. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 121–127, 1985.

[Smi90] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1990.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1988.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

Appendix

The specifications of *Integer knapsack* and *Traveling salesman problem* are reported in this appendix.

Integer 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 $\Sigma_{u \in U} c(u)v(u) \geq K$?

```
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 = 135;
   K = 210;
SPECIFICATION
   IntFunc(U,take_up,0..trunc(B/(min {s(u)})).
         // the max number of copies of any object is B
         // divided by the minimum of the sizes of the objects
   table(Obj,Value,Size) <-- take_up(Obj,Num), data(Obj,V,S),
                             Value == Num * V, Size == Num * S.
   fail <-- SUM(table,(#3) S), S > K.
   fail <-- SUM(table,(#2) V), V < B.
```

Traveling salesman

INSTANCE:	Set C of m cities, distance $d(c_i, c_j) \in Z^+$, for each pair of cities $c_i, c_j \in C$, a positive integer B
QUESTION:	Is there a tour of C having length B or less, i.e., a permutation $\langle c_{\pi(1)}, \ldots c_{\pi(m)} \rangle$ of C such that $\left[\sum_{i=1}^{n-1} d(c_{\pi(i)}), c_{\pi(i+1)})\right] + d(c_{\pi(m)}), c_{\pi(1)}) \leq B$?

```
DATABASE
       C = {london,rome,paris,amsterdam};
       D = {(london,rome,3125), (rome,london,3125),
            (london,paris,1230), (paris,london,1230),
            (london,amsterdam,800), (amsterdam,london,800),
            (rome,paris,2370), (paris,rome,2370),
            (rome,amsterdam,2000), (amsterdam,rome,2000),
            (paris,amsterdam,700), (amsterdam,paris,700)};
            // encoding (ci,cj,d(ci,cj))
       B = 6500;
       M = 4;
SPECIFICATION
       Permutation(C,tour).
       distance(C2,D) <-- tour(C2,1), tour(C1,0), d(C1,C2,D1),
                          M1 == M - 1, tour(C,M1), d(C,C1,D2),
                          D == D1 + D2.
       distance(C,P) <-- tour(C,N), N1 == N - 1, tour(C1,N1),
                         distance(C1,D1), d(C1,C,D), P == D1 + D.
       fail <-- M1 == M - 1, tour(X,M1), distance(X,K), K > B.
```