TLP **7** (1): 1–29, 2006. © 2006 Cambridge University Press
doi:10.1017/S1471068406002857 Printed in the United Kingdom

Combining relational algebra, SQL*, constraint modelling, and local search**

MARCO CADOLI and TONI MANCINI
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy
`cadoli|tmancini@dis.uniroma1.it`

submitted 15 December 2003; revised 15 April 2005; accepted 5 January 2006

Abstract

The goal of this paper is to provide a strong integration between constraint modelling and relational DBMSs. To this end we propose extensions of standard query languages such as relational algebra and SQL, by adding constraint modelling capabilities to them. In particular, we propose non-deterministic extensions of both languages, which are specially suited for combinatorial problems. Non-determinism is introduced by means of a *guessing* operator, which declares a set of relations to have an arbitrary extension. This new operator results in languages with higher expressive power, able to express all problems in the complexity class NP. Some syntactical restrictions which make data complexity polynomial are shown. The effectiveness of both extensions is demonstrated by means of several examples. The current implementation, written in Java using local search techniques, is described.

KEYWORDS: Constraint modelling and programming, relational databases, relational algebra, SQL, local search

1 Introduction

The efficient solution of NP-hard combinatorial problems, such as resource allocation, scheduling, planning, etc. is crucial for many industrial applications, and it is often achieved by means of ad-hoc procedural hand-written programs. Declarative programming languages like AMPL (Fourer *et al.* 1993) and OPL (Van Hentenryck 1999) or libraries (ILOG-98 1998) for expressing constraints are commercially available. Data encoding the instance are either in text files in an ad-hoc format, or in standard relational DBs accessed through libraries callable from programming languages such as C++ (ILOG-DBLINK 1999). In other words, there is not a strong integration between data definition and constraint modelling and programming languages.

* This paper is an extended and revised version of earlier work (Cadoli and Mancini 2002).

Indeed, such an integration is particularly needed in industrial environments, where the necessity for solving combinatorial problems coexists with the presence of large databases where data to be processed lie. Hence, constraint solvers that operate externally to the databases of the enterprise may lead to a series of disadvantages, first of all a potential lack of the integrity of recorded data. To this end, a better coupling between standard data repositories and constraint solving engines is highly desiderable.

The goal of this paper is exactly to integrate constraint modelling and programming into relational database management systems (R-DBMSs). In particular, we show how standard query languages for relational databases can be extended in order to give them constraint modelling and solving capabilities: with such languages, constraint problem specifications can be viewed just like (more complex) queries to standard data repositories. In what follows, we propose extensions of standard query languages such as relational algebra and SQL, that are able to formulate queries defining combinatorial and constraint problems.

In principle relational algebra can be used as a language for testing constraints. As an example, given relations A and B, testing whether all tuples in A are contained in B can be done by computing the relation $A - B$, and then checking its emptiness. Anyway, it must be noted that relational algebra is unfeasible as a language for expressing NP-hard problems, since it is capable of expressing just a strict subset of the polynomial-time queries (Abiteboul *et al.* 1995). As a consequence, an extension is needed.

The proposed generalization of relational algebra is named *NP-Alg*, and it is proven to be capable of expressing all problems in the complexity class NP. We focus on NP because this class contains the decisional version of most combinatorial problems of industrial relevance (Garey and Johnson 1979). *NP-Alg* is relational algebra plus a simple *guessing* operator, which declares a set of relations to have an arbitrary extension. Algebraic expressions are used to express constraints. Several interesting properties of *NP-Alg* are provided: its data complexity is shown to be NP-complete, and for each problem ξ in NP we prove that there is a fixed query that, when evaluated on a database representing the instance of ξ, solves it. Combined complexity is also addressed.

Since *NP-Alg* expresses all problems in NP, an interesting question is whether a query corresponds to an NP-complete or to a polynomial-time problem. We give a partial answer to it, by exhibiting some syntactical restrictions of *NP-Alg* with polynomial-time data complexity.

In the same way, CONSQL (SQL with constraints) is the proposed non-deterministic extension of SQL, the well-known language for querying relational databases (Ullman 1988), having the same expressive power of *NP-Alg*, and supporting also the specification of optimization problems. We believe that writing a CONSQL query for the solution of a combinatorial optimization problem is only moderately more difficult than writing SQL queries for a standard database application. The advantage of using CONSQL is twofold: it is not necessary to learn a completely new language or methodology, and integration of the problem solver with the information system of the enterprise can be done very smoothly. The effectiveness of both *NP-Alg* and CONSQL as constraint modelling languages is demonstrated by showing several queries which specify combinatorial and optimization problems.

The structure of the paper is as follows. Syntax and semantics of *NP-Alg* are introduced in Section 2. Some examples of *NP-Alg* queries for the specification of NP-complete combinatorial problems are proposed in Section 3. Main computational properties of *NP-Alg*, including data and combined complexity, expressive power, and polynomial fragments, are presented in Section 4. Section 5 contains some details of CONSQL and its implementation CONSQL SIMULATOR, as well as the specification of some real-world combinatorial and optimization problems. Finally, Section 6 contains conclusions as well as references to main related work.

2 *NP-Alg*: Syntax and semantics

We refer to a standard definition of relational algebra with the five operators $\{\sigma, \pi, \times, -, \cup\}$ (Abiteboul *et al.* 1995). Other operators such as "$\bowtie$" and "/" can be defined as usual. Attributes (fields) of relations will be denoted either by their names or by their indexes. As an example, given a relation $R(a, b)$, the selection of tuples in R with the same values for the two attributes will be denoted in one of the following forms: $\underset{R.a=R.b}{\sigma}(R)$, $\underset{a=b}{\sigma}(R)$ (since there is no confusion to what relation a and b refer to), $\underset{\$1=\$2}{\sigma}(R)$. As for join conditions, they will have atoms of the form $a = b$ or $a \neq b$ where a is an attribute name (or even index) of the relation on the left of the join symbol, and b one of that on the right. Finally, temporary relations such as $T = algexpr(\ldots)$ will be used to make expressions easier to read. As usual (Chandra and Harel 1980) queries are defined as mappings which are partial recursive and generic, i.e., constants are uninterpreted.

Let D denote a finite relational database, $\mathbf{R}$ the set of its relations, and DOM the unary relation representing the set of all constants occurring in D.

Definition 2.1 (*Syntax of NP-Alg*)
An *NP-Alg* expression has two parts:

1. A set $\mathbf{Q} = \{Q_1^{(a_1)}, \ldots, Q_n^{(a_n)}\}$ of new relations of arbitrary arity, denoted as *Guess* $Q_1^{(a_1)}, \ldots, Q_n^{(a_n)}$. Sets $\mathbf{R}$ and $\mathbf{Q}$ must be disjoint.
2. An ordinary expression *exp* of relational algebra on the new database schema $[\mathbf{Q}, \mathbf{R}]$.

For simplicity, until Section 4 we focus on *boolean queries*, i.e., queries that admit a yes/no answer. For this reason we restrict *exp* to be a relation which we call *FAIL*.

Definition 2.2 (*Semantics of NP-Alg*)
The semantics of an *NP-Alg* expression is as follows:

1. For each possible extension *ext* of the relations in $\mathbf{Q}$ with elements in *DOM*, the relation *FAIL* is evaluated, using ordinary rules of relational algebra.
2. If there exists an extension *ext* such that the expression for *FAIL* evaluates to the empty relation "$\emptyset$" (denoted as $FAIL \diamond \emptyset$), the *answer* to the boolean query is "yes". Otherwise the *answer* is "no".

When the answer is "yes", the extension of relations in $\mathbf{Q}$ is a *solution* for the problem instance.

A trivial implementation of the above semantics obviously requires exponential time, since there are exponentially many possible extensions of the relations in $\mathbf{Q}$. Anyway, as we will show in Section 4.3, some polynomial-time cases indeed exist.

The reason why we focus on a relation named $FAIL$ is that, typically, it is easy to specify a decision problem as a set of constraints (cf. forthcoming Sections 3 and 5). As a consequence, an instance of the problem has a solution if and only if there is an arbitrary choice of the guessed relations such that all constraints are satisfied, i.e., $FAIL = \emptyset$. A $FOUND^{(1)}$ query can be anyway defined as $FOUND = DOM - \pi_{\$1}(DOM \times FAIL)$. In this case, the answer is "yes" if and only if there is an extension ext such that $FOUND \neq \emptyset$.

3 Examples of *NP-Alg* queries

In this section we show the specifications of some NP-complete problems, as queries in *NP-Alg*. All examples are on uninterpreted structures, i.e., on unlabeled directed graphs, because we adopt a pure relational algebra with uninterpreted constants. As a side-effect, the examples show that, even in this limited setting, we are able to emulate bounded integers and ordering. This is very important, because the specification of very simple combinatorial problems requires bounded integers and ordering.

In Section 5 we use the full power of CONSQL to specify some real-world problems.

3.1 Graph k-coloring

We assume a directed graph is represented as a pair of relations $NODES^{(1)}(n)$ and $EDGES^{(2)}(from, to)$ (with tuples in $EDGES^{(2)}$ having components in $NODES^{(1)}$, hence, $DOM = NODES$). A graph is *k-colorable* if there is a k-partition $Q_1^{(1)}, \dots, Q_k^{(1)}$ of its nodes, i.e., a set of k sets such that:

- $\forall i \in [1,k], \forall j \in [1,k], j \neq i \rightarrow Q_i \cap Q_j = \emptyset$,
- $\bigcup_{i=1}^{k} Q_i = NODES$,

and each set Q_i has no pair of nodes linked by an edge. The problem is well-known to be NP-complete for $k \geqslant 3$ (cf., e.g., (Garey and Johnson 1979)), and it can be specified in *NP-Alg* as follows:

$$Guess\ Q_1^{(1)}, \dots, Q_k^{(1)}; \tag{1a}$$

$$FAIL\_DISJOINT = \bigcup_{i \neq j \in \{1,\dots,k\}} Q_i \bowtie Q_j; \tag{1b}$$

$$FAIL\_COVER = NODES\ \Delta \bigcup_{i=1}^{k} Q_i; \tag{1c}$$

$$FAIL\_PARTITION = FAIL\_DISJOINT \cup FAIL\_COVER; \tag{1d}$$

$$FAIL\_COLORING = \pi_{\$1}\left[\bigcup_{i=1}^{k}\left(\left(\sigma_{\$1\neq\$2}(Q_i \times Q_i)\right) \bowtie_{\substack{\$1=EDGES.from\\ \$2=EDGES.to}} EDGES\right)\right]; \tag{1e}$$

$$FAIL = FAIL\_PARTITION \cup FAIL\_COLORING. \tag{1f}$$

Expression (1a) declares k new relations of arity 1. Expression (1f) collects all constraints a candidate coloring must obey to:

- (1b) and (1c) make sure that $Q_1,\ldots,Q_k$ is a partition of *NODES* ("Δ" is the symmetric difference operator, i.e., $A \ \Delta \ B = (A - B) \cup (B - A)$, useful for testing equality since $A \ \Delta \ B = \emptyset \iff A = B$).
- (1e) checks that each set Q_i has no pair of nodes linked by an edge.

As an example, let $k = 3$ and the database be as follows:

NODES

n
1
2
3
4

EDGES

from	*to*
1	2
1	4
2	3

An extension of Q_1, Q_2 and Q_3 such that $FAIL = \emptyset$ is:

Q_1

—
2
4

Q_2

—
1

Q_3

—
3

Note that such an extension constitutes a solution to the coloring problem.

We observe that in the specification above the *FAIL_PARTITION* relation (1d) makes sure that an extension of $Q_1^{(1)},\ldots,\ Q_k^{(1)}$ is a *k-partition* of *NODES*. Such a constraint can be very useful for the specification of problems, so we introduce, as syntactic sugar, an expression:

$$failPartition^{(1)}\big(N^{(k)}, P_1^{(k)},\ldots,P_n^{(k)}\big),$$

which returns an empty relation if and only if $\{P_1^{(k)},\ldots,P_n^{(k)}\}$ is a partition of $N^{(k)}$. The prefix *fail* in the name of the expression reminds the user that it should be used in checking constraints. We note that the arity of *failPartition* can, without loss of generality, be fixed to 1, since we can always project out the remaining columns. Other useful syntactic sugar will be introduced in the following examples, and is summarized in Section 3.5.

3.2 Independent set

Let a (directed) graph be defined, as usual, by the two relations $NODES^{(1)}$ and $EDGES^{(2)}$, and let $k \leqslant |NODES|$ be an integer, which is specified by a relation $K^{(1)}$

containing exactly k tuples. A subset N of $NODES$, with $|N| \geqslant k$ is said to be an *independent set of size at least* k of the graph if N contains no pair of nodes linked by an edge.

The problem of determining whether an input graph has an independent set of size at least k is NP-complete (Garey and Johnson 1979), and it can be easily specified in *NP-Alg*. However, since we have to compare the size of N with the integer k (i.e., with the size of relation K), before presenting the *NP-Alg* query for the Independent set problem, we need a method to compare the size of two relations $N^{(1)}$ and $K^{(1)}$. This can be done by deciding whether a proper *function* that maps tuples in N to tuples in K exists. In particular:

- $|N| = |K|$ if and only if there exists a total bijective function between N and K;
- $|N| \geqslant |K|$ if and only if there exists a partial surjective function from N to K;
- $|N| \leqslant |K|$ if and only if there exists a total injective function from N to K.

To define relational algebra expressions that check whether a relation $FUN^{(d+r)}$ is a (total, injective, surjective, or bijective) *function* from domain $D^{(d)}$ to range $R^{(r)}$, we define the following expressions (for the sake of simplicity, we write definitions for $d = r = 1$, but their extensions to arbitrary d and r are straightforward):

- $\mathit{failFunction}(FUN^{(2)}, D^{(1)}, R^{(1)}) =$

$$\left(\underset{\$1}{\pi}(FUN) - D\right) \cup \left(\underset{\$2}{\pi}(FUN) - R\right) \cup \underset{\$1}{\pi}\left(FUN \underset{\substack{\$1=\$1 \\ \wedge \\ \$2\neq\$2}}{\bowtie} FUN\right),$$

  where the first and second subexpressions check whether tuples in FUN are in the cartesian product $D \times R$, and the third checks whether FUN is mono-valued;
- $\mathit{failTotal}(FUN^{(2)}, D^{(1)}, R^{(1)}) = D - \underset{\$1}{\pi}(FUN)$;
- $\mathit{failSurjective}(FUN^{(2)}, D^{(1)}, R^{(1)}) = R - \underset{\$2}{\pi}(FUN)$;
- $\mathit{failInjective}(FUN^{(2)}, D^{(1)}, R^{(1)}) = FUN \underset{\substack{\$1\neq\$1 \\ \wedge \\ \$2=\$2}}{\bowtie} FUN$.

The above expressions evaluate to the empty relation if and only if relation FUN is, respectively, a function, a total, surjective, or injective relation (in the mathematical sense) from tuples of relation D to tuples of relation R.

By using the above expressions, we can design new ones, with the goal of comparing the size of two relations D and R:

- $\mathit{failGeqSize}(AUX, D, R) = \mathit{failFunction}(AUX, D, R) \cup \mathit{failSurjective}(AUX, D, R)$;
- $\mathit{failLeqSize}(AUX, D, R) = \mathit{failGeqSize}(AUX, R, D)$;
- $\mathit{failEqSize}(AUX, D, R) = \mathit{failLeqSize}(AUX, D, R) \cup \mathit{failGeqSize}(AUX, D, R)$,

where AUX is an auxiliary guessed relation that encodes the function between D and R. Such auxiliary guessed relations will be omitted as arguments in the remainder of the paper if they are not used anywhere else, to enhance readability.

Returning to the example, the following *NP-Alg* query specifies the Independent set problem:

$$\begin{aligned} &\textit{Guess } N^{(1)};\\ &\textit{FAIL} = \textit{failGeqSize}^{(1)}(N,K) \cup \\ &\qquad \pi_{\$1}\left[(N \times N) \underset{\substack{\$1=\textit{EDGES.from}\\ \wedge\\ \$2=\textit{EDGES.to}}}{\bowtie} \textit{EDGES}\right]. \end{aligned}$$

The first subexpression of *FAIL* specifies the constraint $|N| \geqslant k$, while the second one evaluates to the empty relation if and only if no pair of nodes in N is linked by an edge. An extension of N is an independent set (with size at least k) of the input graph if and only if the corresponding *FAIL* relation is empty.

3.3 Clique

Given an undirected graph, i.e., the *EDGES* relation is symmetric, and an integer $k \leqslant |\textit{NODES}|$, a subset N of *NODES*, with $|N| \geqslant k$ is said to be a *clique of size at least k* if every pair of distinct nodes of N is linked by an edge (i.e., the subgraph induced by N is *complete*).

The problem of determining whether a graph has a clique of size at least k is NP-complete (Garey and Johnson 1979), and it can be specified in *NP-Alg* as follows (k is encoded as a relation $K^{(1)}$ with exactly k tuples):

$$\begin{aligned} &\textit{Guess } N^{(1)};\\ &\textit{FAIL} = \textit{failGeqSize}(N,K) \cup \left(\sigma_{\$1\neq\$2}(N \times N) \underset{\substack{\$1=\$1\\ \wedge\\ \$2=\$2}}{\bowtie} \textit{complement}^{(2)}(\textit{EDGES})\right). \end{aligned}$$

The structure of the query is very similar to the one of the previous example, except for the new expression $\textit{complement}^{(k)}(R^{(k)})$, which can be defined as

$$\textit{complement}^{(k)}\big(R^{(k)}\big) = \rho_{\substack{\$1\to R.\$1\\ \vdots\\ \$k\to R.\$k}}(\textit{DOM}^{k} - R),$$

and returns the *active complement* of the relation given as argument (ρ is the field-renaming operator, used to name all columns of the output relation like those of R). Obviously the above query can be written in several other ways. As an example, a more efficient one would use the difference operator, instead of the join; notwithstanding this, we have chosen the above query to show the use of *complement*.

3.4 More examples

We can specify in *NP-Alg* other famous problems over graphs like *Dominating set*, *Transitive closure*, and *Hamiltonian path*. It is worth noting that Transitive closure, indeed a polynomial-time problem, is not expressible in relational algebra (Abiteboul *et al.* 1995), because it intrinsically requires a form of recursion (cf. Section 6). In *NP-Alg* recursion can be simulated by means of guessing. As for Hamiltonian path, this is the problem of finding a traversal of a graph which touches each node exactly once. The possibility to specify the Hamiltonian path problem in *NP-Alg* has interesting consequences which deserve some comments. Consider a unary relation DOM, with $|DOM| = M \neq 0$ and the complete graph C defined by the relations $NODES = DOM$ and $EDGES = DOM \times DOM$. A Hamiltonian path H of C is a total ordering of the M elements in DOM: in fact it is a *successor* relation. The transitive closure of H is the corresponding *less-than* relation. As a consequence, we have the possibility to use *bounded integers* in the range $[1, M]$ in our framework, and also arithmetic operations on them.

Furthermore, the Hamiltonian paths of C correspond to the *permutations* of $[1, M]$. Permutations are very useful for the specification of several problems. As an example, in the *n-queens* problem (in which the goal is to place n non-attacking queens on an $n \times n$ chessboard) a candidate solution is a permutation of order n, representing the assignment of a pair $\langle \text{row}, \text{column} \rangle$ to each queen. Interestingly, to check the attacks of queens on diagonals, in *NP-Alg* we can guess a relation encoding the subtraction of elements in DOM.

Other interesting problems, not involving graphs, can be specified in *NP-Alg*: *Satisfiability of a propositional formula* and *Evenness of the cardinality of a relation* are some examples.

3.5 Useful syntactic sugar

Previous examples show that guessing relations as subsets of DOM^k (for integer k) is enough to express many NP-complete problems. The forthcoming Theorem 4.3 shows that this is indeed enough to express all problems in NP.

Nevertheless, expressions such as *failPartition* can make queries more readable. In this section we briefly summarize the main expressions that we designed.

- $empty^{(1)}(R^{(k)}) = DOM - \underset{\$1}{\pi}(DOM \times R^{(k)})$, evaluates to the empty relation if R is a non-empty one (and vice versa).
- $complement^{(k)}(R^{(k)})$ evaluates to the active complement (with respect to DOM^k) of R (cf. Section 3.3).
- $failPartition^{(1)}(N^{(k)}, P_1^{(k)}, \ldots, P_n^{(k)})$ (cf. Section 3.1) evaluates to the empty relation if and only if $\{P_1^{(k)}, \ldots, P_n^{(k)}\}$ is a partition of N.
- $failSuccessor^{(1)}(SUCC^{(2k)}, N^{(k)})$ evaluates to the empty relation if and only if $SUCC$ encodes a correct successor relation on elements in N, i.e., a 1-1 correspondence with the interval $[1, |N|]$ (essentially by checking whether $SUCC$ is a Hamiltonian path on the graph with edges defined by $N \times N$).
- $failPermutation^{(1)}(PERM^{(2k)}, N^{(k)})$ evaluates to the empty relation if and only if $PERM$ is a permutation of the elements in $N^{(k)}$. The ordering sequence is given by the first k columns of $PERM$.
- $failFunction^{(1)}(FUN^{(d+r)}, D^{(d)}, R^{(r)})$, $failTotal^{(1)}(FUN^{(d+r)}, D^{(d)}, R^{(r)})$, $failInjective^{(1)}(FUN^{(d+r)}, D^{(d)}, R^{(r)})$, $failSurjective^{(1)}(FUN^{(d+r)}, D^{(d)}, R^{(r)})$

(cf. Section 3.2) evaluate to the empty relation if and only if *FUN* is, respectively, a function, a total, injective or surjective relation from tuples in *D* to those in *R*. We remark that, since elements in *R* can be ordered (cf. Section 3.4), *FUN* is also an *integer function* from elements of *D* to the interval $[1, |R|]$. Integer functions are very useful for the specification of *resource allocation* problems, such as *Integer knapsack* (see also examples in Section 5.2).

- $failEqSize^{(1)}(N, K)$, $failGeqSize^{(1)}(N, K)$, $failLeqSize^{(1)}(N, K)$ (cf. Section 3.2) evaluate to the empty relation if and only if $|N|$ is, respectively, $=$, $\geqslant$, $\leqslant$ $|K|$.

4 Computational aspects of *NP-Alg*

In this section we focus on the main computational aspects of *NP-Alg*: data and combined complexity, expressive power, and polynomial fragments.

Technically, the results presented in this section can be easily obtained from corresponding ones formulated for other languages, e.g., existential second order logic (ESO). Nevertheless, we believe that when designing a constraint modelling language it is of fundamental importance, from the methodological point of view, to ascertain its main computational properties.

4.1 Data and combined complexity

The *data complexity*, i.e., the complexity of query answering assuming the database as input and a fixed query (Abiteboul *et al.* 1995), is one of the most important computational aspects of a language, since queries are typically small compared to the database.

Since we can express some NP-complete problems in *NP-Alg* (cf. Section 3), the problem of deciding whether $FAIL \diamond \emptyset$ is NP-hard. Moreover we can prove that the data complexity for such a problem is in NP by using the following argument. It is possible to generate, in non-deterministic polynomial time, an extension *ext* of *Q*. The answer is “yes” if and only if there is such an *ext* that makes $FAIL = \emptyset$. The last check, being the evaluation of an ordinary relational algebra expression, can be done in polynomial time in the size of the database. The above considerations give us the first computational result on *NP-Alg*.

Theorem 4.1
The data complexity of deciding whether $FAIL \diamond \emptyset$ for an *NP-Alg* query, where the input is the database, is NP-complete.

Another interesting measure is *combined complexity*, where both the database and the query are part of the input. It is well known that, typically, the combined complexity of a language is much higher than its data complexity (Vardi 1982). As for *NP-Alg*, it is possible to show that, when both the database and the query are part of the input, the problem of determining whether $FAIL \diamond \emptyset$ is hard for the complexity class NE, defined as $\bigcup_{c>1} NTIME\ (2^{cn})$ (Papadimitriou 1994), i.e., the class of all problems solvable by a non-deterministic machine in time bounded by 2^{cn}, where n is the size of the input and c is an arbitrary constant.

Theorem 4.2
The combined complexity of deciding whether $FAIL \diamond \emptyset$ for an *NP-Alg* query, where the input is both the database and the query, is NE-hard.

The proof is quite long and is delayed to Section Appendix A.

4.2 Expressive power

The *expressiveness* of a query language characterizes the problems that can be expressed as fixed, i.e., instance independent, queries. In this section we prove the main result about the expressiveness of *NP-Alg*, by showing that it captures exactly NP, or equivalently (Fagin 1974) queries in the existential fragment of second-order logic (ESO).

Of course it is very important to be assured that we can express *all* problems in the complexity class NP. In fact, Theorem 4.1 says that we are able to express *some* problems in NP. We remind that the expressive power of a language is less than or equal to its data complexity. In other words, there exist languages whose data complexity is hard for class C in which not every query in C can be expressed; several such languages are known (Abiteboul *et al.* 1995).

In order to show that *NP-Alg* is able to express all problems in NP, we illustrate a method that transforms an arbitrary formula in ESO into a *NP-Alg* query. We remind that, by Fagin's theorem (Fagin 1974), any collection $\mathbf{D}$ of finite databases over $\mathbf{R}$ is NP-recognizable if and only if it can be defined by a existential second order formula. In particular, we deal with ESO formulae of the following kind:

$$(\exists \mathbf{S})\ (\forall \mathbf{X})\ (\exists \mathbf{Y})\ \varphi(\mathbf{X}, \mathbf{Y}), \tag{2}$$

where φ is a first-order formula (without quantifiers) containing variables among $\mathbf{X}, \mathbf{Y}$ and involving relational symbols in $\mathbf{S} \cup \mathbf{R} \cup \{=\}$. The reason why we can restrict our attention to second-order formulae in the above normal form is explained in (Kolaitis and Papadimitriou 1991). As usual, "$=$" is always interpreted as "identity".

The transformation works in two steps:

1. The first-order formula $\varphi(\mathbf{X}, \mathbf{Y})$ obtained by eliminating all quantifiers from (2) is translated into an expression *PHI* of plain relational algebra;
2. The query ψ is defined as:

$$\begin{aligned} &Guess\ Q_1^{(a_1)}, \ldots, Q_n^{(a_n)};\\ &FAIL = DOM^{|\mathbf{X}|} - \pi_{\mathbf{X}}(PHI), \end{aligned} \tag{3}$$

where $a_1, \ldots, a_n$ are the arities of the n predicates in $\mathbf{S}$, and $|\mathbf{X}|$ is the number of variables occurring in $\mathbf{X}$.

The first step is rather standard (Abiteboul *et al.* 1995), and is briefly sketched here just to give the intuition. A relation R (with the same arity) is introduced for each predicate symbol r in the relational vocabulary of φ, i.e., $\mathbf{R} \cup \mathbf{S}$. An atomic formula of first-order logic is translated as the corresponding relation, possibly prefixed by a selection that accounts for constant symbols and/or repeated variables, and by a renaming of attributes mapping the arguments. Selection can be used also for dealing with atoms involving equality. Inductively, the relation corresponding to a complex first-order formula is built as follows:

- $f \wedge g$ translates into $F \bowtie G$, where F and G are the translations of f and g, respectively;
- $f \vee g$ translates into $F' \cup G'$, where F' and G' are derived from the translations F and G to account for the (possibly) different schemata of f and g;
- $\neg f(\mathbf{Z})$ translates into $\underset{\substack{\$1 \to F.\$1 \\ \vdots \\ \$|\mathbf{Z}| \to F.\$|\mathbf{Z}|}}{\rho} (DOM^{|\mathbf{Z}|} - F)$ (ρ is the column renaming operator, needed to name columns of $(DOM^{|\mathbf{Z}|} - F)$ like those of F).

It is worth noting that a better translation avoids the insertion of occurrences of the DOM relation for the important class of *safe formulae* (Abiteboul *et al.* 1995). However, these issues are out of the scope of this paper, and will not be taken into account.

Relations obtained through such a translation will be called *q-free*, because they do not contain the projection operator (that plays the role of an existential *quantification*), except those implicit in equi-joins. Intuitively, this means that there are no existential quantifiers.

The following theorem claims that the above translation is correct.

Theorem 4.3
For any NP-recognizable collection $\mathbf{D}$ of finite databases over $\mathbf{R}$ – characterized by a formula of the kind (2) – a database D is in $\mathbf{D}$, i.e., $D \models (\exists \mathbf{S})(\forall \mathbf{X})(\exists \mathbf{Y})\, \varphi(\mathbf{X}, \mathbf{Y})$, if and only if $FAIL \diamond \emptyset$, when ψ (cf. formula (3)) is evaluated on D.

Proof
(Only if part) If $D \in \mathbf{D}$, it follows that an extension Σ for predicates in $\mathbf{S}$ exists, such that:

$$[D, \Sigma] \models (\forall \mathbf{X})(\exists \mathbf{Y})\ \varphi(\mathbf{X}, \mathbf{Y}).$$

By translating φ on the right side into relational algebra (according to the point 1 above), we obtain a relational expression PHI on the relational vocabulary given by relations corresponding to predicates in $\mathbf{R}$, plus those corresponding to predicates in $\mathbf{S}$ (i.e., relations in $\mathbf{Q}$).

When evaluating PHI on the new database $[D, \Xi]$, where Ξ are the extensions of relations in $\mathbf{Q}$ corresponding to the extensions of predicates in Σ, we obtain that for all tuples $\langle \mathbf{X} \rangle$ there exists a tuple $\langle \mathbf{Y} \rangle$ such that the tuple $\langle \mathbf{X}, \mathbf{Y} \rangle$ belongs to PHI, i.e.:

$$\forall \langle \mathbf{X} \rangle \exists \langle \mathbf{Y} \rangle : \langle \mathbf{X}, \mathbf{Y} \rangle \in PHI.$$

Since $\langle \mathbf{X} \rangle \in DOM^{|\mathbf{X}|}$, we obtain that $DOM^{|\mathbf{X}|} \subseteq \underset{\mathbf{X}}{\pi}(PHI)$, implying that the expression for $FAIL$ in the *NP-Alg* query (3) evaluates to the empty relation for the extension Ξ of the guessed tables $\mathbf{Q}$.

(If part) Suppose that $D \notin \mathbf{D}$. This implies that $D \models \neg(\exists \mathbf{S})(\forall \mathbf{X})(\exists \mathbf{Y})\varphi(\mathbf{X}, \mathbf{Y})$ or, equivalently, that:

$$D \models (\forall \mathbf{S})(\exists \mathbf{X})(\forall \mathbf{Y})\ \neg\varphi(\mathbf{X}, \mathbf{Y})$$

By translating formula φ into relational algebra, we obtain that, for every extension Ξ of relations in $\mathbf{Q}$ (corresponding to predicates in $\mathbf{S}$), there exists at least one tuple $\langle \mathbf{X} \rangle$ such that for every tuple $\langle \mathbf{Y} \rangle$, tuple $\langle \mathbf{X}, \mathbf{Y} \rangle$ does not belong to *PHI*. This implies that:

$$\exists \langle \mathbf{X} \rangle \in DOM^{|\mathbf{X}|} : \langle \mathbf{X} \rangle \notin \underset{\mathbf{X}}{\pi}(PHI),$$

and so that the expression for *FAIL* in the *NP-Alg* query (3) does not evaluate to the empty relation for all possible extensions Ξ of the guessed tables $\mathbf{Q}$. □

4.3 Polynomial fragments

Polynomial fragments of second-order logic have been presented (Gottlob *et al.* 2004). In this section we use some of those results to show that it is possible to isolate polynomial fragments of *NP-Alg*.

Theorem 4.4

Let s be a positive integer, *PHI* a q-free expression of relational algebra over the relational vocabulary $edb(D) \cup \{Q^{(s)}\}$, and Y_1, Y_2 the names of two attributes of *PHI*. An *NP-Alg* query of the form:

$$\begin{aligned} &Guess\ Q^{(s)};\\ &FAIL\ = (DOM \times DOM) - \underset{Y_1,Y_2}{\pi}(PHI). \end{aligned}$$

can be evaluated in polynomial time in the size of the database.

Proof

This class of *NP-Alg* queries corresponds to the *Eaa* prefix class of second-order logic described in (Gottlob *et al.* 2004), which is proved to be polynomial by a mapping into instances of 2SAT. The correctness of the translation is formally guaranteed by Theorem 4.3. □

Some interesting queries obeying the above restriction can indeed be formulated. As an example, *2-coloring* can be specified as follows (when $k = 2$, k-coloring, cf. Section 3.1, becomes polynomial):

$$\begin{aligned} &Guess\ C^{(1)};\\ &FAIL = DOM \times DOM - \begin{bmatrix} complement(EDGES) \cup & \\ C \times complement(C) \cup complement(C) \times C \end{bmatrix}. \end{aligned}$$

C and its complement denote the 2-partition. The constraint states that each edge must go from one subset to the other one.

Another polynomial problem of this class is *2-partition into cliques* (cf., e.g., (Garey and Johnson 1979)), which amounts to decide whether there is a 2-partition of the nodes of a graph such that the two induced subgraphs are complete. An *NP-Alg* query which specifies the problem is:

$$\begin{aligned}&\mathit{Guess}\ P^{(1)};\\ &\mathit{FAIL} = \mathit{DOM} \times \mathit{DOM} - \\ &\qquad [\mathit{complement}(P) \times P \ \cup\ P \times \mathit{complement}(P) \ \cup\ \mathit{EDGES}].\end{aligned}$$

A second polynomial class is defined by the following theorem.

Theorem 4.5
Let $\mathit{PHI}(X_1,\ldots,X_k,Y_1,Y_2)$ $(k > 0)$ be a q-free expression of relational algebra over the relational vocabulary $\mathit{edb}(D) \cup \{Q^{(1)}\}$. An *NP-Alg* query of the form:

$$\begin{aligned}&\mathit{Guess}\ Q^{(1)};\\ &X(X_1,\ldots,X_k) = \mathit{PHI}(X_1,\ldots,X_k,Y_1,Y_2) \ / \ \underset{\substack{\$1\to Y_1\\ \$2\to Y_2}}{\rho}(\mathit{DOM} \times \mathit{DOM});\\ &\mathit{FAIL} \ = \mathit{empty}(X).\end{aligned}$$

can be evaluated in polynomial time in the size of the database.

Proof
This class of *NP-Alg* queries corresponds to the E_1e^*aa prefix class of second-order logic (Gottlob *et al.* 2004), which, in turn, is proved to be polynomial by a mapping into 2SAT. Also in this case, the correctness of the translation is formally guaranteed by Theorem 4.3. □

As an example, the specification for the *Graph disconnectivity* problem, i.e., to check whether a graph is not connected, belongs to this class.

5 The conSql language

In this section we describe the conSql language, a non-deterministic extension of SQL (able to express also optimization problems) whose optimization-free subset has the same expressive power as *NP-Alg*, and present some specifications written in this language.

5.1 Syntax of conSql

conSql is a strict superset of SQL. The problem instance is described as a set of ordinary tables, using the data definition language of SQL. The novel construct `CREATE SPECIFICATION` is used to define a problem specification. It has three parts, two of which correspond to the parts of Definition 2.1:

1. Definition of the guessed tables, by means of the new keyword `GUESS`;
2. Optional definition of an objective function, by means of one of the two keywords `MAXIMIZE` and `MINIMIZE`;
3. Specification of the constraints that must be satisfied by guessed tables, by means of the standard SQL keyword `CHECK`.

Furthermore, the user can specify the desired output by means of the new keyword RETURN. In particular, the output is computed when an extension of the guessed tables satisfying all constraints and such that the objective function is optimized is found. Of course, it is possible to specify many guessed tables, constraints and returned tables. The syntax is as follows (we write it in BNF, with terminals either capitalized or quoted, and, for every terminal or non-terminal *a*, "[*a*]" meaning optionality, "*a*∗" a list of an arbitrary number of *a*, and "*a*+" meaning "*a*(*a*∗)"):

```
CREATE SPECIFICATION problem_name '('
       (GUESS TABLE table_name ['('aliases')'] AS guessed_table_spec)+
       ((MAXIMIZE | MINIMIZE) '('aggregate_query')'
       (CHECK '(' condition ')')+
       (RETURN TABLE return_table_name AS query)*
')'
```

The guessed table table_name gets its schema from its definition guessed_table_spec. The latter expression is similar to a standard SELECT-FROM-WHERE SQL query, except for the FROM clause that can contain also expressions such as:

```
SUBSET OF SQL_from_clause |
[TOTAL | PARTIAL] FUNCTION_TO '(' (range_table | min '..' max) ')'
      AS field_name_list OF SQL_from_clause |
(PARTITION '(' n ')' | PERMUTATION) AS field_name OF SQL_from_clause
```

with SQL_from_clause being the content of an ordinary SQL FROM clause (e.g., a list of tables). The schema of such expressions consists in the attributes of SQL_from_clause, plus the extra field_name (or field_name_list), if present.

In the FROM clause the user is supposed to specify the shape of the search space, either as a plain subset (like in *NP-Alg*), or as a mapping (i.e., partition, permutation, or function) from the domain defined by SQL_from_clause. Mappings require the specification of the range and the name of the extra field(s) containing range values. As for PERMUTATION, the range is implicitly defined to be a subset of integers. As for FUNCTION_TO the range can be either an interval min..max of a SQL enumerable type, (e.g., integers) or the set of values of the primary key of a table denoted by range_table. The optional keyword PARTIAL means that the function can be defined over a subset of the domain (the default is TOTAL). We remind the reader that using partitions, permutations or functions does not add any expressive power to the language (cf. Section 3.5).

As for the objective function, the user is supposed to specify a query whose output is a monadic table with only one tuple of an SQL totally ordered type (e.g., integers or reals), typically by making use of SQL aggregate operators like COUNT, SUM, etc.

It is possible to specify constraints on the guessed tables by using ordinary SQL boolean conditions, e.g., EXISTS, NOT EXISTS, IN, NOT IN, =ANY, =ALL, etc.

Finally, the query that defines a returned table is an ordinary SQL query on the tables defining the problem instance plus the guessed ones, and it is evaluated for an arbitrary extension of the guessed tables encoding an optimal solution. This is consistent with the semantics adopted by all state-of-the-art systems for Constraint Programming.

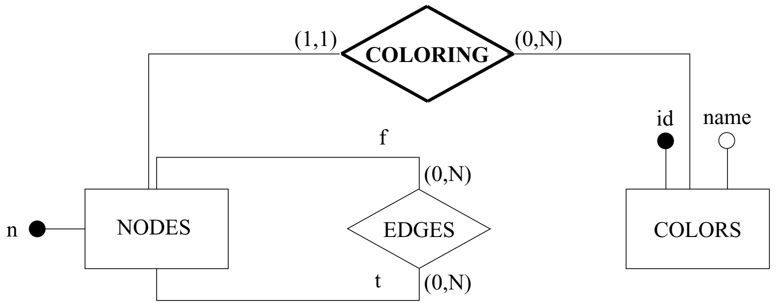

Fig. 1. ER diagram of the database schema for the k-coloring problem. The guessed table *COLORING* is in boldface.

Once a problem has been specified, its solution can be obtained with an ordinary SQL query on the return tables:

```
SELECT field_name_list
FROM problem_name.return_table_name
WHERE condition
```

The table `ANSWER(n INTEGER)` is implicitly defined locally to the `CREATE SPECIFICATION` construct, and it is empty if and only if the problem has no solution.

5.2 Examples

In this subsection we exhibit the specification of some problems in CONSQL. In particular, to highlight its similarity with *NP-Alg*, we show the specification of the graph coloring problem of of Section 3.1. Afterwards, we exploit the full power of the language and show how some real-world problems can be easily specified. In all the examples, we describe the schema of the input database, and underline key fields.

5.2.1 Graph k-coloring

We assume an input database over the schema shown in the Entity-Relationship (ER) diagram in Figure 1, thus containing relations *NODES(n)*, *EDGES(f,t)* (encoding the graph), and *COLORS(id,name)* (listing the k colors). Once a database (i.e., a problem instance) has been created (by using standard SQL commands), a CONSQL specification of the k-coloring problem is the following:

```
CREATE SPECIFICATION Graph_Coloring (
   /* COLORING contains tuples of the kind <NODES.n, COLORS.id>,
      with COLORS.id arbitrarily chosen. */
 GUESS TABLE COLORING AS
   SELECT n, color FROM TOTAL FUNCTION_TO(COLORS) AS color OF NODES
    CHECK ( NOT EXISTS (
      SELECT * FROM COLORING C1, COLORING C2, EDGES
      WHERE C1.n <> C2.n AND C1.color = C2.color
        AND C1.n = EDGES.f AND C2.n = EDGES.t ))
    RETURN TABLE SOLUTION AS SELECT COLORING.n, COLORS.name
      FROM COLORING, COLORS WHERE COLORING.color = COLORS.id
)
```

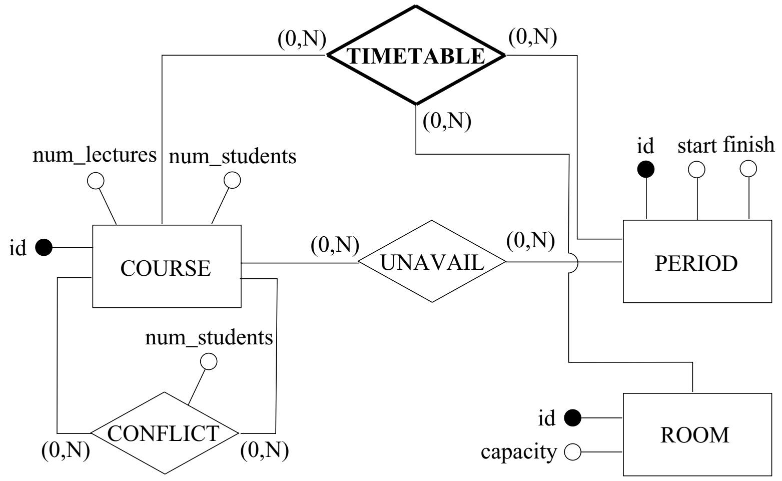

Fig. 2. ER diagram of the database schema for the University course timetabling problem. The guessed table *TIMETABLE* is in boldface.

The GUESS part of the problem specification defines a new (binary) table COLORING, with fields n and color, as a total function from the set of NODES to the set of COLORS. The CHECK statement expresses the constraint an extension of COLORING table must satisfy to be a solution to the problem, i.e., there are no two distinct nodes linked by an edge which are assigned the same color.

The RETURN statement defines the output of the problem by a query that is evaluated for an extension of the guessed table that satisfies every constraint. The user can ask for such a solution with the statement

```
SELECT * FROM Graph_Coloring.SOLUTION
```

As described in the previous subsection, if no coloring exists, the system table Graph_Coloring.ANSWER will contain no tuples. This can be easily checked by the user, in order to obtain only a significant Graph_Coloring.SOLUTION table.

5.2.2 University course timetabling

The *University course timetabling* problem (Schaerf 1999) consists in finding the weekly scheduling for all the lectures of a set of university courses in a given set of classrooms. We consider a variant of the original problem in which the objective function to minimize is the total number of students that have to attend overlapping lectures.

The input database schema is shown in Figure 2, and consists of the following relations:

- *COURSE(<u>id</u>, num_lectures, num_students)*, consisting of tuples $\langle c, l, s\rangle$ meaning that the course c needs l lectures a week, and has s enrolled students.
- *PERIOD(<u>id</u>, start, finish)* encoding (non-overlapping) periods, plus information on start and finish time.

- *ROOM(id, capacity)*. A tuple $\langle r, c\rangle$ means that room r has capacity c.
- *CONFLICT(course1, course2, num_students)*. A tuple $\langle c1, c2, n\rangle$ means that courses *c1* and *c2* have n common students.
- *UNAVAIL(course, period)*. A tuple $\langle c, p\rangle$ means that the teacher of course c is not available for teaching at period p.

A solution to the problem is a (guessed) relation *TIMETABLE(period, room, course)* with tuples $\langle p, r, c\rangle$ meaning that at period p in room r there is a lecture of course c. If for some values of the *room* and *period* fields there is no tuple in the relation *TIMETABLE*, then the room is unused in that period.

A CONSQL specification of the timetabling problem, given an input database, is the following:

```
CREATE SPECIFICATION University_Timetabling (
  GUESS TABLE TIMETABLE(period, room, course) AS
    SELECT p.id, r.id, course
    FROM PARTIAL FUNCTION_TO(COURSE) AS course OF PERIOD p, ROOM r
  // Objective function
  MINIMIZE ( SELECT SUM(c.num_students)
    FROM TIMETABLE t1, TIMETABLE t2, CONFLICT c
    WHERE t1.period = t2.period AND t1.course <> t2.course AND
          c.course1 = t1.course AND c.course2 = t2.course
  )
  // At most one lecture of a course per period
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t1, TIMETABLE t2
    WHERE t1.course = t2.course AND
          t1.period = t2.period AND t1.room <> t2.room
  ))
  // Unavailability constraints
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t, UNAVAIL u
    WHERE t.course = u.course AND t.period = u.period
  ))
  // Capacity constraints
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t, COURSE c, ROOM r
    WHERE t.course = c.id AND t.room = r.id AND
          c.num_students > r.capacity
  ))
  // Teaching requirements
  CHECK ( NOT EXISTS (
    SELECT * FROM COURSE c
    WHERE c.num_lectures <>
      ( SELECT COUNT(*) FROM TIMETABLE t
          WHERE t.course = c.id
      )
  ))
  RETURN TABLE SOLUTION AS SELECT * FROM TIMETABLE
)
```

In particular, the constraints force extensions of the guessed table *TIMETABLE* to be such that:

- There is at most one lecture of a course per period, i.e., there cannot be two different rooms allocated for the same course in the same time slot;
- Unavailability constraints are respected, i.e., no lecture is scheduled in a period for which the relevant teacher is unavailable;
- Capacity constraints are respected, i.e., no room is allocated for courses having a number of students that exceeds its capacity;
- Teaching requirements are satisfied, i.e., all courses have a room and a time slot assigned for all the lectures they need.

An extension for guessed table *TIMETABLE* that satisfies the constraints above is an optimal solution to the University course timetabling problem if it minimizes the overall number of students that are expected to attend conflicting lectures, i.e., lectures that are scheduled at the same time.

5.2.3 Aircraft landing

The *aircraft landing* problem (Beasley *et al.* 2000) consists in scheduling landing times for aircraft. Upon entering within the radar range of the air traffic control (ATC) at an airport, a plane requires a *landing time* and a *runway* on which to land. The landing time must lie within a specified time window, bounded by an *earliest time* and a *latest time*, depending on the kind of the aircraft. Each plane has a most economical, preferred speed. A plane is said to be assigned its *target time*, if it is required to fly in to land at its preferred speed. If ATC requires the plane to either slow down or speed up, a cost incurs. The bigger the difference between the assigned landing time and the target landing time, the bigger the cost. Moreover, the amount of time between two landings must be greater than a specified minimum (the *separation time*) that depends on the planes involved. Separation times depend on the aircraft landing on the same or different runways (in the latter case they are smaller).

Our objective is to find a landing time for each planned aircraft, encoded in a guessed relation *LANDING*, satisfying all the previous constraints, and such that the total cost (i.e., the sum of the costs associated with each aircraft) is minimized. The input database schema is shown in Figure 3, and consists of the following relations:

- $AIRCRAFT(\underline{id}, target\_time, earliest\_time, latest\_time, bef\_cost, aft\_cost)$, listing aircraft planned to land, together with their target times and landing time windows; the cost associated with a delayed or advanced landing at time x is given by $bef\_cost \cdot \max[0, t - x] + aft\_cost \cdot \max[0, x - t]$, where t is the aircraft target time.
- $RUNWAY(\underline{id})$ listing all the runways of the airport.
- $SEPARATION(\underline{first}, \underline{second}, int\_same\_rw, int\_diff\_rw)$. A tuple $\langle a, a', is, id\rangle$ means that if aircraft a' lands after aircraft a, then landing times must be separated by is (resp. id) minutes if they land on the same runway (resp. on different runways).

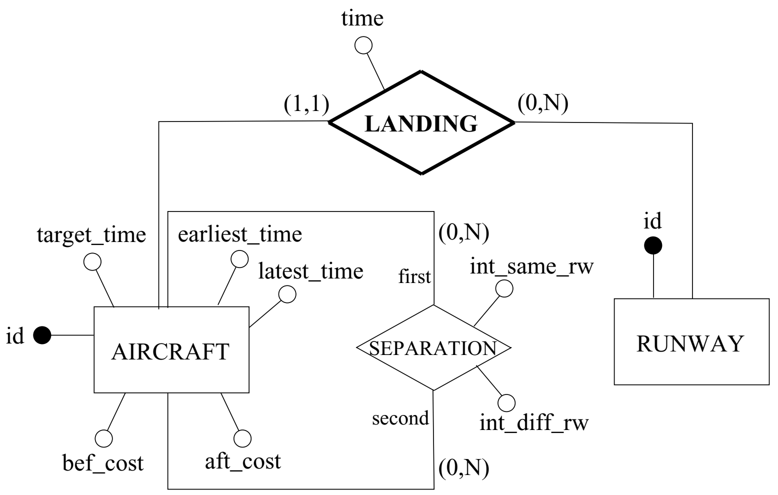

Fig. 3. ER diagram of the database schema for the Aircraft landing problem. The guessed table *LANDING* is in boldface.

In the following specification, the search space is a total function assigning an aircraft to a landing time and a runway. For the sake of simplicity, landing times are expressed in minutes after a conventional time instant, e.g., the scheduling starting time, and the time horizon is set to one day, i.e., 24 × 60 minutes.

```
CREATE SPECIFICATION Aircraft_Landing (
  GUESS TABLE LANDING AS
    SELECT ar.id AS aircraft, ar.runway, at.time
    FROM (TOTAL FUNCTION_TO(RUNWAY) AS runway OF AIRCRAFT) ar,
         (TOTAL FUNCTION_TO(0..24*60-1) AS time OF AIRCRAFT) at
    WHERE ar.id = at.id
   // Objective function
   MINIMIZE ( SELECT SUM(cost)
     FROM (
       SELECT a.id, (a.bef_cost * (a.target_time - l.time)) AS cost
         FROM AIRCRAFT a, LANDING l
         WHERE a.id = l.aircraft AND l.time <= a.target_time
       UNION // advanced plus delayed aircraft
       SELECT a.id, (a.aft_cost * (l.time - a.target_time)) AS cost
         FROM AIRCRAFT a, LANDING l
         WHERE a.id = l.aircraft AND l.time > a.target_time
     ) AIRCRAFT_COST // Contains tuples <aircraft, cost>
   )
   // Time window constraints
   CHECK ( NOT EXISTS (
     SELECT * FROM LANDING l, AIRCRAFT a WHERE l.aircraft = a.id
       AND ( l.time > a.latest_time OR l.time < a.earliest_time )
   ))
   // Separation constraints
   CHECK ( NOT EXISTS (
     SELECT * FROM LANDING l1, LANDING l2, SEPARATION sep
     WHERE l1.aircraft <> l2.aircraft AND l1.time <= l2.time AND
```

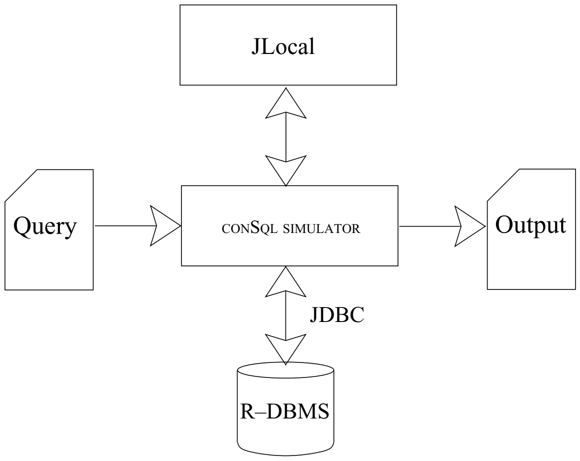

Fig. 4. conSql simulator overall architecture.

```
        sep.first = l1.aircraft AND sep.second = l2.aircraft AND
        (  ( (l1.runway = l2.runway) AND
             (l2.time - l1.time) < sep.int_same_rw ) OR
           ( (l1.runway <> l2.runway) AND
             (l2.time - l1.time) < sep.int_diff_rw )
    )))
    RETURN TABLE SOLUTION AS SELECT * FROM LANDING
  )
```

In particular, the constraints force extensions of the guessed table *LANDING* to be such to respect both time window constraints (i.e., the actual landing time for each aircraft must lie inside its landing time window), and separation constraints (encoded in the *SEPARATION* relation). Such an extension is an optimal solution to the Aircraft landing problem if it minimizes the overall cost.

5.3 conSql simulator

conSql simulator is an application that works as an interface to a traditional R-DBMS. It simulates the behavior of a conSql server by reading from its input stream conSql queries, i.e., ordinary sql queries and commands, and problem specifications. Ordinary sql queries and commands are simply passed to the underlying R-DBMS, while problem specifications are processed. The overall architecture of the system is depicted in Figure 4. In particular, `CREATE SPECIFICATION` constructs are parsed, creating the new tables (corresponding to the guessed ones) and an internal representation of the search space. The search space is then explored by the solver, looking for an element corresponding to an optimal solution, by posing appropriate queries to the R-DBMS (in standard sql). As soon as an optimal solution is found, results of the queries specified in the `RETURN` statements are accessible to the user as output.

The implementation of conSql simulator gives much attention to software engineering aspects and to different quality factors of software artifacts. In particular,

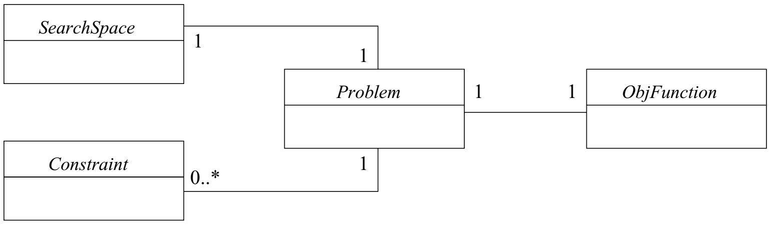

Fig. 5. Portion of the conceptual UML class diagram for the language-independent problem modelling module.

the system is platform independent and highly portable, since it is written in Java, and uses the standard JDBC protocol for the connection with the R-DBMS, and the whole architecture presents a neat separation among the language parser, the problem modelling module and the solver engine (JLOCAL), so as to emphasize qualities such as modularity, extendability and reusability. In particular, the problem modelling module allows to represent problem specifications in a *language independent* fashion, relying on abstract concepts such as *Problem*, *Search space*, *Objective function*, and *Constraint*, as the conceptual UML diagram in Figure 5 shows. In this way, the solving engine JLOCAL, interacting with the abstract problem modelling module, is independent of the particular language, i.e., CONSQL. The only language-dependent part of the system is the parser for the CONSQL language, that provides concrete implementations for the abstract concepts that compose the problem modelling module, building the internal representation of the problem instance, the search space, and the constraints, and for some of the services needed by the solver.

As for the search methods, the solver engine JLOCAL exploits local search techniques to find solutions. Local search is considered one of the most attractive techniques for solving combinatorial optimization problems (cf., e.g., (Aarts and Lenstra 1997)), being able to solve instances of realistic size in reasonable time. Several local search algorithms have been implemented in JLOCAL, among them Hill climbing and Tabu search (Glover and Laguna 1997), and several strategies that combine different solvers for doing the search are present (e.g., Tandem search, in which two different solvers are used in sequence). Additional local search strategies can be simply added by subclassing the *LocalSearchSolver* class (not described here for the sake of simplicity).

It is worth noting that the user is completely unaware of the search techniques implemented by the system. In particular, definitions for neighborhoods, moves, aspiration functions, etc., are made by CONSQL SIMULATOR itself, starting from the types of the guessed tables defined in the specification (i.e., subsets, functions, permutations, partitions).

The development of CONSQL SIMULATOR has been done according to the iterative model of the software life-cycle. In particular, three iterations were expected. The first, prototypical, version of the system, which was used only to test specifications, relied on a purely enumerative approach, and, of course, no considerations on performances could be done. As for the present version, which is at the second iteration of the development process, we added the local search engine, but the connection with the DBMS is yet completely black box. In particular, JLocal uses the DBMS both for maintaining the current state, for checking constraints, and for evaluating which neighbor to visit next. The main motivation behind this choice, is that current DBMSs offer means to answer queries efficiently, especially in case of very large instances. Nonetheless, since constraints are evaluated from scratch in every visited state, performances cannot be good, and only instances of small sizes can be actually solved.

In the third version of the system, which is currently under development, we are adding the following functionalities:

1. The ability of checking constraints incrementally: constraints' check is the main source of inefficiency of the current version, since they are evaluated from scratch for every visited state, and for all its neighbors, in order to choose the best move. Hence, the number of queries posted to the DBMS is very high, and all of them are answered independently from each other. However, it is clear that, when using local search technology, only a small variation in the number of constraint violations is expected, when moving from one state to its neighbors. To this end, our goal is to make the DBMS able to compute only variations to constraints' violations when performing checks. This is expected to greatly increase the overall performances of the system, since we can rely on very sophisticated algorithms to, e.g., maintain and synchronize views, actually present in currently available DBMSs.
2. The use of a much more complex local search engine. In particular, we are currently integrating EasyLocal++ (Di Gaspero and Schaerf 2003), a very sophisticated solver, with our system. This can lead to better algorithms, and to a fine tuning of their parameters.
3. The addition of an optional "search" part in `CREATE SPECIFICATION` constructs, as it happens in, e.g., OPL, in order to provide the user with the possibility of declaring which search algorithm to adopt, as well as the types of neighborhoods, moves, aspiration functions, etc. Of course, as already claimed, our goal is to provide good defaults for the options in this part, as, e.g., OPL does, by letting the system able to automatically make a good choice for these issues, depending on the specification at hand.
4. To provide a better coupling with a particular open-source DBMS, in order to make the system able to directly use the DBMS' APIs, instead of interacting by means of (inefficient, but highly portable) protocols like JDBC.

6 Conclusions, related and future work

In this paper we have tackled the issue of strong integration between constraint modelling and programming and up-to-date technology for storing data. In particular we have proposed constraint languages which have the ability to interact with data repositories in a standard way. To this end, we have presented *NP-Alg*, an extension of relational algebra which is specially suited for combinatorial problems. The main feature of *NP-Alg* is the possibility of specifying, via a form of non-determinism, a set of relations that can have an arbitrary extension. This allows the specification of a search space suitable for the solution of combinatorial problems, with ordinary relational algebra expressions defining constraints. Although *NP-Alg* provides just a very simple guessing operator, many useful search spaces, e.g., permutations and functions, can be defined as syntactic sugar.

Several computational properties of *NP-Alg* have been shown, including data and combined complexity, and expressive power. Notably, the language is shown to capture exactly all the problems in the complexity class NP, which includes many combinatorial problems of industrial relevance. In the same way, we have proposed CONSQL, a non-deterministic extension of SQL, with the same expressive power of *NP-Alg*, which is suitable also for specifying optimization problems. The effectiveness of *NP-Alg* and CONSQL both as complex query and constraint modelling languages has been demonstrated by showing several queries which specify combinatorial problems.

Other extensions of relational algebra have already been proposed. The most important examples are the languages $Alg + while$ and $Alg + while^+$, where, respectively, a non-inflationary and an inflationary fixpoint semantics is added (Abiteboul *et al.* 1995). Both these languages are capable of expressing the Transitive Closure query, but have very different expressive power: $Alg + while$ can express queries in PSPACE, but the language captures exactly this class only on ordered databases (i.e., databases in which a total order among all constants occurring in it is fixed). As for $Alg + while^+$ instead, it can express only polynomial-time queries, and the language captures the whole PTIME class only on ordered databases. A feature for expressing linear recursion has recently been added also to SQL (SQL'99), by means of the `WITH` construct. However, both the aforementioned extensions of relational algebra, and the new version of SQL do not make such languages suitable for expressing constraint problems.

Several languages and systems for constraint programming are nowadays available either as research and commercial packages. Some of them are in the form of frameworks and libraries. As an example, in ECL^iPS^e (Eclipse) or SICSTUS (Sicstus) a traditional programming language such as PROLOG is enhanced by means of libraries and specific constructs for specifying constraints, which are then solved by highly optimized algorithms. The ILOG Optimization suite (ILOG-98 1998) provides instead libraries for expressing constraints callable by host general-purpose programming languages like C++.

Specification languages natively developed for constraint modelling and programming are also available, either commercially like OPL (Van Hentenryck 1999) and AMPL (Fourer *et al.* 1993) or as research prototypes, like ESRA (Flener *et al.* 2004), all of them offering an ad-hoc syntax for problem specifications. Similarly to *NP-Alg* and CONSQL, they support a clear distinction between the data and the problem description level, but differently from them, *NP-Alg* and CONSQL use standard and well-known languages for specifying problem specifications, that are considered just like queries over a relational database representing the input instance. We believe that this feature allows for a wider diffusion of the declarative constraint modelling paradigm in industrial environments, permitting a very strong integration with the information system of the enterprise. Conversely, the other systems usually get input data from text files in ad-hoc formats, and additional machinery is needed to build such files from the content of a relational database, and for storing problem solutions. Even if some of them have plug-ins that can be used to make connections to databases, e.g., (ILOG-DBLINK 1999), data are always processed outside the DBMS, hence leading to a potential lack of data integrity.

Several query languages capable of capturing the complexity class NP have been shown in the literature. As an example, in (Kolaitis and Papadimitriou 1991) an extension of datalog (the well-known recursive query language (Ullman 1988)) allowing negation is proved to have such a property. Another extension of datalog capturing NP, without negation but with a form of non-determinism, is proposed in (Cadoli and Palopoli 1998). Other rule-based languages with different semantics have also been proposed: SMODELS (Simons *et al.* 2002) which relies on stable models semantics, and DLV (Leone *et al.*) which is based on answer set programming. They also are based on negation and recursion. On the other hand, *NP-Alg* captures NP without recursion. Actually, recursion can be simulated by non-determinism, and it is possible to write, e.g., the transitive closure query in *NP-Alg*. Being non-recursive, *NP-Alg* is more similar to plain existential second order logic. Nevertheless, it retains the functional character of relational algebra, which sometimes makes it easier (with respect to rule-based languages) to specify a problem.

For what concerns CONSQL, we believe it is a clear step towards a language for both declarative constraint modelling and complex queries to relational databases, which relies on standard and well-known technologies. Currently, the most adopted solution for evaluating complex queries over a relational database is to embed SQL into a general-purpose programming language, like Java or C++, thus by processing stored data and intermediate results outside the database. CONSQL instead has been designed for being implemented *inside* the DBMS, so guaranteeing all transactional properties to the query evaluation process.

As for the proposed implementation of CONSQL SIMULATOR, it is conceived to be based on a purely declarative language and to be ready to use, i.e., it does not require any additional code to be written by the user. Other systems for local search do, however, exist, either in forms of declarative languages for modelling in a concise way local search algorithms (cf., e.g., (Michel and Van Hentenryck 2000; Van Hentenryck and Michel 2003)) or, alternatively, in forms of libraries or frameworks (cf., e.g., Local++ (Schaerf *et al.* 2000)), hence providing algorithms that rely on additional application-specific code provided by the user. CONSQL SIMULATOR is different from such systems in that it provides the user with the ability of modelling an optimization problem by means of a language, i.e., CONSQL, that is completely unaware of the particular solving technology used. It is responsibility of the engine to provide the local search solver with all the information needed to explore the search space (e.g., description of neighborhoods, moves, etc.). This choice is currently made starting from the types of the guessed tables defined in the specification, and future work has to be done in order to better exploit the different alternatives, as discusses at the end of Subsection 5.3.

CONSQL SIMULATOR will be released as free and potentially open source software, thus allowing the system to receive improvements and extensions from the community.

Appendix A Combined complexity of *NP-Alg*

In this section we prove Theorem 4.2. The proof consists in reducing an NE-complete problem, *Succint* 3-*coloring* (Kolaitis and Papadimitriou 1991), i.e., the "succinct version" of the graph 3-coloring problem, into an *NP-Alg* query. It is worth noting that the resulting *NP-Alg* query is *not uniform* with respect to the problem instance, but this is exactly what the definition of *combined complexity* (as opposed to *data complexity*) states. The Succint 3-coloring problem is defined as follows:

Definition Appendix A.1 (*The Succint* 3-*coloring problem*)
Nodes of the input graph are elements of $\{0,1\}^n$, and, instead of an explicitly given *EDGES* relation, there is a *boolean circuit* with $2n$ inputs and one output, such that the value output by the circuit is 1 if and only if the inputs are two n-tuples that encode a pair of nodes connected by an edge. A boolean circuit is a finite set of triples $\{(a_i, b_i, c_i), i = 1, \ldots, k\}$, where $a_i \in \{OR, AND, NOT, IN\}$ is the kind of the gate, and $b_i, c_i < i$ are the inputs of the gate (hence, the whole circuit is acyclic), unless the gate is an input gate ($a_i = IN$), in which case, say, $b_i = c_i = 0$. For NOT gates, $b_i = c_i$. Given values in $\{0,1\}$ for the input gates, we can compute the values of all gates one by one by starting from the first one. The value of the circuit is the value of the last gate. Finally, the *Succint* 3-*coloring problem* is the following: Given a boolean circuit with $2n$ inputs and one output, is the graph thus presented 3-colorable?

The Succint 3-coloring problem is proven to be NE-complete in the same paper (Kolaitis and Papadimitriou 1991).

Reduction of Succint 3-*coloring into an NP-Alg query.* Given an input boolean circuit $G = \{g_i = (a_i, b_i, c_i) \mid 1 \leqslant i \leqslant k\}$ with $2n$ inputs and one output, we construct the *NP-Alg* query ψ that specifies the Succint 3-coloring problem (on the graph represented by circuit G) as follows.

As for the set $\mathbf{Q}$ of guessed relations, we declare a relation $G_i^{(2n)}$ for every gate i, i.e., for every triple $g_i = (a_i, b_i, c_i)$, $(1 \leqslant i \leqslant k)$. Moreover, we declare in $\mathbf{Q}$ three more relations, $COL_1^{(n)}$, $COL_2^{(n)}$, $COL_3^{(n)}$, encoding the partition of the nodes into 3 groups, analogously to the specification for k-coloring given in Section 3.1. So, the *Guess* part of the *NP-Alg* query being built is the following:

$$Guess\ G_1^{(2n)}, \ldots, G_k^{(2n)}, COL_1^{(n)}, COL_2^{(n)}, COL_3^{(n)};$$

Intuitively, relations G_i will contain all tuples $\langle \mathbf{X}, \mathbf{Y} \rangle$, with $\mathbf{X} = \langle X_1, \ldots, X_n \rangle$, and $\mathbf{Y} = \langle Y_1, \ldots, Y_n \rangle$ (i.e., binary encodings of the nodes X and Y) for all pairs of nodes X and Y that make the output of the i-th gate 1.

The expression for *FAIL* is of the following kind:

$$FAIL = FAIL\_CIRCUIT \cup FAIL\_PARTITION \cup FAIL\_COLORING.$$

The first subexpression evaluates to the empty relation if and only if the guessed extension for the G_i relations correctly encodes the circuit, while the second and the third ones evaluate to the empty relation if and only if relations COL_1, COL_2, COL_3, are a partition of the graph nodes and a correct coloring of the graph (we omit their definitions, since they are very similar to those presented in Section 3.1).

The expression for *FAIL_CIRCUIT* contains in turn one of the following subexpressions $FAIL\_G_i$, $1 \leqslant i \leqslant k$, for every gate i, according to its type a_i. In particular:

- If $a_i = AND$, then $FAIL\_G_i = G_i \,\Delta\, [G_{b_i} \cap G_{c_i}]$;
- If $a_i = OR$, then $FAIL\_G_i = G_i \,\Delta\, [G_{b_i} \cup G_{c_i}]$;
- If $a_i = NOT$, then $FAIL\_G_i = G_i \,\Delta\, \left[DOM_{01}^{2n} - G_{b_i}\right]$;
- If $a_i = IN$, then $FAIL\_G_i = G_i \,\Delta\, \sigma_{\$j=1}(DOM_{01}^{2n})$, assuming that the i-th gate (of type IN) is the j-th input of the circuit.

In the above definition, we used the relation DOM_{01}, defined as:

$$DOM_{01} = \sigma_{\substack{\$1 \neq AND \,\wedge \\ \$1 \neq OR \,\wedge \\ \$1 \neq NOT \,\wedge \\ \$1 \neq IN}} (DOM)$$

that will contain at most the two tuples $\langle 0 \rangle$ and $\langle 1 \rangle$ (since *DOM* would also contain constants for the gate types). Thus, the expression for *FAIL_CIRCUIT* is the following:

$$FAIL\_CIRCUIT = \bigcup_{i=1}^{k} FAIL\_G_i.$$

It remains to prove that the expression for *FAIL_CIRCUIT* evaluates to the empty relation if and only if guessed relations $G_1, \ldots, G_k$ correctly encode the boolean circuit representing the input graph, i.e., if and only if for all i, relation G_i contains exactly all $2n$-tuples (encoding pairs of nodes given as input to the circuit) that make the output of the i-th gate 1. This is what the following lemma claims.

Lemma Appendix A.1
Let $G = \{g_i = (a_i, b_i, c_i) \mid 1 \leqslant i \leqslant k\}$ be a boolean circuit encoding a graph, and let ψ be the *NP-Alg* query built as described above. An extension for guessed tables $G_1^{(2n)}, \ldots, G_k^{(2n)}$ exists such that the expression for *FAIL_CIRCUIT* evaluates to the empty relation. Moreover, for such an extension, each G_i contains exactly all $2n$-tuples $\langle X_1, \ldots, X_n, Y_1, \ldots, Y_n \rangle$ that make the output of the i-th gate 1. As a consequence, the extension for G_k contains all $2n$-tuples that encode pairs of nodes linked by an edge.

Proof
We first show that, if extensions for $G_1, \ldots G_k$ in the *NP-Alg* query ψ exist that make the expression for *FAIL_CIRCUIT* evaluate to the empty relation (by making all

the expressions for $FAIL\_G_i$ evaluate to the empty relation), then, for every input $\{X_1,\dots,X_n,Y_1,\dots,Y_n\}$ to the circuit, each gate i ($1 \leqslant i \leqslant k$) outputs 1 if and only if the $2n$-tuple $\langle X_1,\dots,X_n,Y_1,\dots,Y_n\rangle$ belongs to the corresponding G_i. Secondly, we show that such an extension indeed exists. The proof of the first point is by induction on the index i:

$i = 1$: Gate g_1 is, by construction, of type IN (i.e., $a_1 = IN$). Let us assume that g_1 is the j-th input to the circuit, i.e., its output is 1 if and only if the j-th input to the circuit is 1. As it can be observed from the definition of $FAIL\_G_1$, since by hypothesis it evaluates to the empty relation, G_1 contains all $2n$-tuples that have 1 as the j-th component.

$i > 1$: Let us assume that the lemma holds for all i' such that $1 \leqslant i' < i$, and let us consider the i-th gate (of type $a_i \in \{IN, AND, OR, NOT\}$) and the extension for the corresponding guessed relation G_i. Since, by hypothesis, $FAIL\_G_i$ evaluates to the empty relation, it can be easily observed by its definition that:

- If $a_i = IN$, assuming that g_i is the j-th input to the circuit, G_i contains, by construction, all $2n$-tuples that have 1 as the j-th component;
- If $a_i = AND$, it follows by induction that G_{b_i} and G_{c_i} contain exactly those tuples that make the output of, respectively, gates g_{b_i} e g_{c_i} 1. By construction, the extension for G_i contains exactly those tuples that belong to both G_{b_i} and G_{c_i}.
- If $a_i = OR$ an analogous argument holds, showing that G_i contains exactly those tuples that belong to G_{b_i} or to G_{c_i}.
- If $a_i = NOT$, it follows by induction that G_{b_i} (in this case $b_i = c_i$) contains exactly those tuples that make the output of gate g_{b_i} 1. By construction, the extension for G_i contains exactly those tuples in DOM_{01}^{2n} that do not belong to G_{b_i}.

As for the second point of the proof, it is easy to show that an extension for $G_1,\dots G_k$ that makes all the expressions for $FAIL\_G_i$ evaluate to the empty relation indeed exists. The key observation is that expressions for $FAIL\_G_i$ essentially define which tuples must belong to each G_i (more precisely, each $FAIL\_G_i$ evaluates to the empty relation if and only if G_i contains exactly the tuples that belong to the relational algebra expression on the right of the "Δ" symbol), and that the *Guess* part of the query generates all possible extensions of those relations with elements in $DOM \supset DOM_{01}$. $\square$

Lemma Appendix A.1 claims that an extension for $G_1,\dots G_k$ in query ψ that makes the expression for $FAIL\_CIRCUIT$ evaluate to the empty relation exists, and is the one that correctly models the boolean circuit representing the input graph. It remains to prove that the whole query ψ is such that $FAIL \diamond \emptyset$ if and only if the input graph is 3-colorable. This is claimed by the following result:

Lemma Appendix A.2
Let $G = \{g_i = (a_i, b_i, c_i) \mid 1 \leqslant i \leqslant k\}$ be a boolean circuit encoding a graph, and let ψ be the *NP-Alg* query built as described above. The expression for *FAIL* in ψ evaluates to the empty relation for a given extension of $G_1,\dots G_k, COL_1, COL_2, COL_3$ if and

only if $G_1, \ldots G_k$ correctly encode the circuit G and COL_1, COL_2, COL_3 represent a valid coloring of the input graph. Thus, $FAIL \diamond \emptyset$ if and only if the input graph is 3-colorable.

Proof
The boolean circuit G is translated into k guessed relations $G_1, \ldots G_k$. The correctness of the translation is claimed by Lemma Appendix A.1. Moreover, the *Guess* part of query ψ generates also all possible extensions for three more guessed relations, i.e., COL_1, COL_2, COL_3. As discussed in Section 3.1, the expression for $FAIL\_PARTITION \cup FAIL\_COLORING$ evaluates to the empty relation if and only if COL_1, COL_2, COL_3 define a valid coloring of the graph. □

From previous lemmas, it follows the proof of Theorem 4.2 that states the combined complexity of *NP-Alg*:

Proof of Theorem 4.2
Immediate, from Lemma Appendix A.2 and from the NE-completeness of the Succint 3-coloring problem (Kolaitis and Papadimitriou 1991). □

Acknowledgements

This research has been supported by MIUR (Italian Ministry for Instruction, University, and Research) under the FIRB project ASTRO (Automazione dell'Ingegneria del Software basata su Conoscenza), and under the COFIN project "Design and development of a software system for the specification and efficient solution of combinatorial problems, based on a high-level language, and techniques for intensional reasoning and local search". Special thanks are due to the anonymous reviewers, whose comments and suggestions lead to significant improvements of this paper.

References

Aarts, E. and Lenstra, J. K. 1997. *Local search in combinatorial optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester.

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases.* Addison, Reading, Massachussetts.

Beasley, J. E., Krishnamoorthy, M., Sharaiha, Y. M. and Abramson, D. 2000. Scheduling aircraft landings – the static case. *Transportation Science 34*, 180–197.

Cadoli, M. and Mancini, T. 2002. Combining Relational Algebra, SQL, and Constraint Programming. *Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2002).* Lecture Notes in Artificial Intelligence, vol. 2309. Springer, Santa Margherita Ligure, Genova, Italy, 147–161.

Cadoli, M. and Palopoli, L. 1998. Circumscribing datalog: expressive power and complexity. *Theoretical Computer Science 193*, 215–244.

Chandra, A. and Harel, D. 1980. Computable queries for relational databases. *Journal of Computer and System Sciences 21*, 156–178.

Di Gaspero, L. and Schaerf, A. 2003. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software – Practice and Experience 33,* 8, 733–765.

Eclipse. *ECL^{i}PS^e* Home page. `www-icparc.doc.ic.ac.uk/eclipse/`.

Fagin, R. 1974. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, R. M. Karp, Ed. American Mathematical Society, 43–74.

Flener, P., Pearson, J. and Ågren, M. 2004. Introducing ESRA, a relational language for modelling combinatorial problems. *Proceedings of International Symposium LOPSTR 2003: Revised selected papers*. Lecture Notes in Computer Science, vol. 3018. Springer, Uppsala, Sweden, 214–232.

Fourer, R., Gay, D. M. and Kernigham, B. W. 1993. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson.

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

Glover, F. and Laguna, M. 1997. *Tabu search*. Kluwer Academic, Boston, USA.

Gottlob, G., Kolaitis, P. G. and Schwentick, T. 2004. Existential second-order logic over graphs: Charting the tractability frontier. *Journal of the ACM 51,* 2, 312–362.

ILOG-98 1998. ILOG optimization suite – white paper. Available at `www.ilog.com`.

ILOG-DBLINK 1999. ILOG DBLink 4.1 Tutorial. Available at `www.ilog.com`.

Kolaitis, P. G. and Papadimitriou, C. H. 1991. Why not negation by fixpoint? *Journal of Computer and System Sciences 43*, 125–144.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G. and Perri, Simona abd Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*. To appear.

Michel, L. and Van Hentenryck, P. 2000. Localizer. *Constraints 5,* 1, 43–84.

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

Schaerf, A. 1999. A survey of automated timetabling. *Artificial Intelligence Review 13,* 2, 87–127.

Schaerf, A., Cadoli, M. and Lenzerini, M. 2000. LOCAL++: A C++ framework for local search algorithms. *Software – Practice and Experience 30,* 3, 233–257.

Sicstus. SICStus Prolog home page. `http://www.sics.se/sicstus/`.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2, 181–234.

Ullman, J. D. 1988. *Principles of Database and Knowledge Base Systems*. Vol. 1. Computer Science Press.

Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*. The MIT Press.

Van Hentenryck, P. and Michel, L. 2003. Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*. Lecture Notes in Computer Science, vol. 2833. Springer, Kinsale, Ireland, 65–80.

Vardi, M. Y. 1982. The complexity of relational query languages. *Proceedings of the Fourteenth ACM Symposium on Theory of Computing (STOC'82)*. ACM Press, 137–146.