Combining Relational Algebra, SQL, and Constraint Programming

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

Abstract. The goal of this paper is to provide a strong interaction between constraint programming and relational DBMSs. To this end we propose extensions of standard query languages such as relational algebra (RA) and SQL, by adding constraint solving 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 languages is demonstrated by means of several examples.

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 hand-written programs. Specialized programming languages [7, 15] or libraries [10] 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++ (cf., e.g., [11]). In other words, there is not a strong integration between the data definition and the constraint programming languages.

The goal of this paper is to integrate constraint programming into relational database management systems (R-DBMSs): to this end we propose extensions of standard query languages such as relational algebra (RA) and SQL, by adding constraint solving capabilities to them.

In principle RA 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 RA 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 (cf., e.g., [1]). As a consequence, an extension is indeed needed.

The proposed generalization of RA 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 [8]. *NP-Alg* is RA 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 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, NP-SQL is the proposed non-deterministic extension of SQL, the well-known language for querying relational databases [14], having the same expressive power of *NP-Alg*. We believe that writing an NP-SQL query for the solution of a combinatorial problem is only moderately more difficult than writing SQL queries for a standard database application. The advantage of using NP-SQL 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 NP-SQL as constraint modeling languages is demonstrated by showing several queries which specify combinatorial problems.

2 *NP-Alg*: Syntax and semantics

We refer to a standard definition of RA with the five operators $\{\sigma, \pi, \times, -, \cup\}$ [1]. Other operators such as "$\bowtie$" and "/" can be defined as usual. Temporary relations such as $T = algexpr(\dots)$ will be used to make expressions easier to read. As usual queries are defined as mappings which are partial recursive and generic, i.e., constants are uninterpreted.

Let D denote a finite relational database, $edb(D)$ the set of its relations, and DOM the unary relation representing the set of all constants occurring in D.

Definition 1 (Syntax of *NP-Alg*). *An NP-Alg expression has two parts:*

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

For simplicity, in this paper we focus on *boolean queries*. For this reason we restrict *exp* to be a relation which we call $FAIL$.

Definition 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 RA.*
2. *If there is an extension ext such that* $FAIL = \emptyset$*, the* answer *to the boolean query is "yes" (denoted as* $FAIL \diamond \emptyset$*). 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 iff 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 $FOUND = DOM - \underset{\$1}{\pi}(DOM \times FAIL)$. In this case, the answer is "yes" iff 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 RA with uninterpreted constants. As a side-effect, the examples show that, even in this limited setting, we are able to emulate integers and ordering. This is very important, because the specification of very simple combinatorial problems requires integers and ordering.

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

3.1 *k*-colorability

We assume a directed graph is represented as a pair of relations $NODES^{(1)}$ and $EDGES^{(2)}(from, to)$ ($DOM = NODES$). A graph is *k-colorable* if there is a k-partition $Q_1^{(1)}, \ldots, Q_k^{(1)}$ 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 (cf., e.g., [8]), and it can be specified in *NP-Alg* as follows:

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

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

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

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

$$FAIL\_COLORING = \underset{\$1}{\pi}\left[\bigcup_{i=1}^{k}\left(\left(\underset{\$1 \neq \$2}{\sigma}(Q_i \times Q_i)\right)_{\substack{\$1=EDGES.from \\ \$2=EDGES.to}}\bowtie EDGES\right)\right]; \tag{1e}$$

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

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 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.

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 an expression can be very useful for the specification of problems, so we introduce a *metaexpression*:

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

which returns an empty relation iff $\{P_1^{(k)}, \ldots, P_n^{(k)}\}$ is a partition of $N^{(k)}$. The prefix $fail$ in the name of the metaexpression reminds that it should be used in checking constraints. Other metaexpressions will be introduced in the following examples, and are summarized in Section 3.4.

3.2 Independent set

Let a (directed) graph be defined, as usual, with the two relations $NODES^{(1)}$ and $EDGES^{(2)}$, and let $k \leq |NODES|$ be an integer, which is specified by a relation $K^{(1)}$ containing exactly k tuples. A subset N of $NODES$, with $|N| \geq k$ 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 (cf., e.g., [8]), and it can be easily specified in *NP-Alg*. However, since we have to "count" the elements of N, before presenting the

NP-Alg query for the independent set problem, we show a method to determine whether two relations $N^{(1)}$ and $K^{(1)}$ have the same cardinality or not. Consider the following *NP-Alg* query:

$$
\begin{aligned}
&Guess\ NK^{(2)};\\
&FAIL = \left(\underset{\$1}{\pi}(NK)\ \Delta\ N\right)\ \cup\ \left(\underset{\$2}{\pi}(NK)\ \Delta\ K\right)\ \cup\\
&\qquad \underset{\$1}{\pi}\left(NK \underset{\substack{\$1\neq\$1\\ \wedge\\ \$2=\$2}}{\bowtie} NK\right)\ \cup\ \underset{\$1}{\pi}\left(NK \underset{\substack{\$1=\$1\\ \wedge\\ \$2\neq\$2}}{\bowtie} NK\right).
\end{aligned}
$$

The idea is to guess a binary relation NK which is a bijection between N and K. The first (resp. second) subexpression discards all candidates such that the first (resp. second) column is not the same as N (resp. K). The two joins ($\bowtie$) make sure that exactly one N value is paired to exactly one K value (and vice versa). As a consequence, $FAIL \diamond \emptyset$ iff N and K have the same cardinality. Obviously, deleting the first (resp. second) join, $FAIL \diamond \emptyset$ iff $|N| \geq |K|$ (resp. $|N| \leq |K|$).

Given the reusability of the previous expression, we define the metaexpressions $failSameSize^{(1)}(N, K)$, $failGeqSize^{(1)}(N, K)$, $failLeqSize^{(1)}(N, K)$ as shortcuts for the respective definitions. So, an *NP-Alg* query that specifies the independent set problem is the following:

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

The former subexpression of $FAIL$ specifies the constraint $|N| \geq k$ (to enhance readability, the guessing of the NK relation, used only by the metaexpression, is omitted). The latter one returns an empty relation iff 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 iff the corresponding $FAIL$ relation is empty.

3.3 More examples

We can specify in *NP-Alg* other famous problems over graphs like *dominating set*, *transitive closure* (TC), and *Hamiltonian path* (HP). We remind that TC, indeed a polynomial-time problem, is not expressible in RA (cf., e.g., [1]), because it intrinsically requires a form of recursion. In *NP-Alg* recursion can be simulated by means of guessing.

HP is the problem of finding a traversal of a graph which touches each node exactly once. The possibility to specify HP in *NP-Alg* has some 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 $EDGES = DOM \times DOM$. An HP 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, considering a bijection between the M elements in DOM and the subset $[1, M]$ of the integers, we actually have the possibility to "count" between 1 and M. Furthermore, the Hamiltonian paths of C correspond to the *permutations* of $[1, M]$. Once the elements in DOM have been ordered (so we can consider them as integers), we can introduce arithmetic operations.

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 $\langle$row, 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.

Finally, in the full paper we show the specification of other problems not involving graphs, such as *satisfiability of a propositional formula* and *evenness of the cardinality of a relation.*

3.4 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. Forthcoming Theorem 3 shows that this is indeed enough to express all problems in NP.

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

- $empty^{(1)}(R) = DOM - \pi_{\$1}(DOM \times R^{(k)})$, returns an empty relation if R is a non-empty one (and vice versa).
- $complement^{(k)}(R^{(k)})$ returns the active complement (wrt DOM^k) of R.
- $failPartition^{(1)}(N^{(k)}, P_1^{(k)}, \ldots, P_n^{(k)})$ (cf. Subsection 3.1) returns an empty relation iff $\{P_1^{(k)}, \ldots, P_n^{(k)}\}$ is a partition of N.
- $failSuccessor^{(1)}(SUCC^{(2k)}, N^{(k)})$ returns an empty relation iff $SUCC$ encodes a correct successor relation on elements in N, i.e., a 1-1 correspondence with the interval $[1, |N|]$.
- $failSameSize^{(1)}(N, K)$, $failGeqSize^{(1)}(N, K)$, $failLeqSize^{(1)}(N, K)$ (cf. Subsection 3.2) return an empty relation iff $|N|$ is, respectively, $=$, $\geq$, $\leq$ $|K|$. We remark that a relation NK satisfying $failGeqSize^{(1)}(N, K)$ is actually a *function* with domain N and range K. Since elements in K can be ordered (cf. Subsection 3.3), NK is also an *integer function* from elements of N to the interval $[1, |K|]$. Integer functions are very useful for the specification of *resource allocation* problems, such as *integer knapsack* (see also examples in Section 5.2). In the full paper we show that we can guess general

functions (total, partial, injective, surjective) from a given domain to a given range.
- $failPermutation^{(1)}(PERM^{(2k)}, N^{(k)})$ returns an empty relation iff $PERM$ is a permutation of the elements in N. The ordering sequence is given by the first k columns of $PERM$.

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.

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 (cf. [1]), is one of the most important computational aspects of a language, since queries are typically small compared to the database.

Since we can express NP-complete problems in *NP-Alg* (cf. Section 3), the problem of deciding whether $FAIL\diamond\emptyset$ is NP-hard. Since the upper bound is clearly NP, we have the first computational result on *NP-Alg*.

Theorem 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 possible to show that, in this case, to determine whether $FAIL\diamond\emptyset$ is hard for the complexity class NE defined as $\bigcup_{c>1} NTIME\ (2^{cn})$ (cf. [13]), i.e., the class of all problems solvable by a nondeterministic machine in time bounded by 2^{cn}, where n is the size of the input and c is an arbitrary constant.

Theorem 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.*

In the full paper the theorem is proved by reducing the NE-complete problem of the *succinct 3-colorability* of a graph [12] into the problem of deciding if an *NP-Alg* query has a solution.

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 (cf. [6]) queries in the existential fragment of second-order logic ($SO_\exists$).

Of course it is very important to be assured that we can express *all* problems in the complexity class NP. In fact, Theorem 1 says that we are able to express *some* problems in NP. We remind that the expressive power of a language is in general 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, cf., e.g., [1].

In the following, σ denotes a fixed set of relational symbols not including equality "=", and $\mathbf{S}$ denotes a list of variables ranging over relational symbols distinct from those in σ. By Fagin's theorem [6] any NP-recognizable collection $\mathbf{D}$ of finite databases over σ is defined by a second-order existential formula. In particular, we deal with second-order 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 containing variables among $\mathbf{X}, \mathbf{Y}$ and involving relational symbols in $\sigma \cup \mathbf{S} \cup \{=\}$. The reason why we can restrict our attention to second-order formulae in the above normal form is explained in [12]. As usual, "=" is always interpreted as "identity".

We illustrate a method that transforms a formula of the kind (2) into an *NP-Alg* expression ψ. 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 RA;
2. the expression ψ is defined as:

$$Guess\ Q_1^{(a_1)}, \ldots, Q_n^{(a_n)};\ \ FAIL = DOM^{|\mathbf{X}|} - \underset{\mathbf{X}}{\pi}(PHI), \tag{3}$$

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, and is briefly sketched here just for future reference. A relation R (with the same arity) is introduced for each predicate symbol $r \in \sigma \cup \mathbf{S}$. 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 $\underset{\substack{\$1 \to F.\$1 \\ \vdots \\ \$|\mathbf{Z}| \to F.\$|\mathbf{Z}|}}{\rho} (DOM^{|\mathbf{Z}|} - F)$.

Relations obtained through such a translation will be called *q-free*.

The following theorem claims that the above translation is correct.

Theorem 3. *For any NP-recognizable collection* $\mathbf{D}$ *of finite databases over* σ *–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*.*

4.3 Polynomial fragments

Polynomial fragments of second-order logic have been presented in, e.g., [9]. In this section we use some of those results to show that it is possible to isolate polynomial fragments of *NP-Alg*.

Theorem 4. *Let* s *be a positive integer,* PHI *a q-free expression of RA on a 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:*

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

can be evaluated in polynomial time.

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

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

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., [8]), 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* expression which specifies the problem is:

$$Guess\ P^{(1)};$$
$$FAIL = DOM \times DOM - [complement(P) \times P\ \cup\ P \times complement(P)\ \cup\ EDGES]\,.$$

A second polynomial class (in which, e.g., the *disconnectivity* problem, i.e., to check whether a graph is not connected, can be expressed) is defined by the following theorem.

Theorem 5. *Let $PHI(X_1, \ldots, X_k, Y_1, Y_2)$ $(k > 0)$ be a q-free expression of RA on a relational vocabulary $edb(D) \cup \{Q^{(1)}\}$. An NP-Alg query of the form:*

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

can be evaluated in polynomial time.

The classes identified by the above theorems correspond respectively to the Eaa and $E_1 e^* aa$ classes of [9], which are proved to be polynomial by a mapping into instances of 2SAT.

5 The NP-SQL language

In this section we describe the NP-SQL language, a non-deterministic extension of SQL having the same expressive power as *NP-Alg*, and present some specifications written in this language.

5.1 Syntax of NP-SQL

NP-SQL 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 PROBLEM` is used to specify a problem. It has two parts, which correspond to the two parts of Definition 1:

1. definition of the guessed tables, by means of the new keyword `GUESS`;
2. 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 is found. Of course, it is possible to specify many guessed tables, constraints and return tables. The syntax is as follows (terminals are either capitalized or quoted):

```
CREATE PROBLEM problem_name '('
       (GUESS TABLE table_name ['('aliases')'] AS guessed_table_spec)+
       (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 which 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 that using partitions, permutations or functions does not add any expressive power to the language (cf. Section 3.4.)

Finally, the `query` that defines a return table is an ordinary SQL query on the tables defining the problem instance plus the guessed ones, and it is evaluated for an extension of the guessed tables satisfying all constraints.

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 cond
```

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

5.2 Examples

In this section we exhibit the specification of some problems in NP-SQL. In particular, to highlight its similarity with *NP-Alg*, we show the specification of the graph coloring problem of Section 3.1. Afterwards, we exploit the full power of the language and show how some real-world problems can be easily specified.

***k*-colorability** We assume an input database containing relations `NODES(n)`, `EDGES(f,t)` (encoding the graph), and `COLORS(id,name)` (listing the k colors).

```
CREATE PROBLEM 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.c = C2.c
```

```
        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
)
```

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 of the problem, i.e., not two distinct nodes linked by an edge 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 which 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.

Aircraft landing The *aircraft landing* problem [2] 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*) which 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 is less than or equal to a given threshold. The input database 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.
- $SEPARATION(\underline{i}, \underline{j}, interval, \underline{same\_runway})$ ($same\_runway$ is a boolean field specifying whether aircraft i and j land on the same runway or not). A tuple $\langle i, j, int, s \rangle$ means that if aircraft j lands after aircraft i, then

landing times must be separated by *int*. There are two such values, for $same\_runway = 0$ and 1, respectively. The relation contains a tuple for all combinations of i, j, and $same\_runway$.
- $MAXCOST(c)$, containing just one tuple, the total cost threshold.

In the following specification, the search space is a total function which assigns an aircraft to a landing time (minutes after midnight) and a runway.

```
CREATE PROBLEM Aircraft_Landing (
  GUESS TABLE LANDING(aircraft, runway, time) AS
    SELECT a1.id, runway, time
    FROM TOTAL FUNCTION_TO(RUNWAY) AS runway OF AIRCRAFT a1,
         TOTAL FUNCTION_TO(0..24*60-1) AS time OF AIRCRAFT a2
    WHERE a1.id = a2.id

   // 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 sep.i = l1.aircraft AND
       sep.j = l2.aircraft AND (l2.time - l1.time) < sep.interval)
       OR (l1.time > l2.time AND sep.i = l2.aircraft AND
         sep.j = l1.aircraft AND (l1.time - l2.time) < sep.interval))
         AND (( l1.runway = l2.runway AND sep.same_runway = 1 )
           OR ( l1.runway <> l2.runway AND sep.same_runway = 0 )
   )))
   // Cost constraint
   CHECK ( NOT EXISTS (
     SELECT * FROM MAXCOST WHERE MAXCOST.c < (
       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
         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
       ) AIRCRAFT_COST // Contains tuples <aircraft, cost>
   )))
   RETURN TABLE SOLUTION AS SELECT * FROM LANDING
)
```

5.3 NP-SQL SIMULATOR

NP-SQL SIMULATOR is an application written in Java, which works as an interface to a traditional R-DBMS. It simulates the behavior of an NP-SQL server by

reading an input text file containing a problem specification (in the NP-SQL language), and looking for a solution.

`CREATE PROBLEM` constructs are parsed, creating the new tables (corresponding to the guessed ones) and an internal representation of the search space; ordinary SQL statements, instead, are sent directly to the DBMS. The search space is explored, looking for an element corresponding to a solution, by posing appropriate queries to the R-DBMS (set so as to work in main memory). As soon as a solution is found, the results of the query specified in the `RETURN` statements are accessible to the user.

In the current implementation, used mainly to check correctness of specifications, a simple-minded enumeration algorithm is used to explore the search space. In the future, we plan to perform the exploration by performing a translation of the problem specification to a third party constraint programming system or to an instance of the propositional satisfiability problem. The latter approach has indeed been proven to be promising in [4].

6 Conclusions, related and future work

In this paper we have tackled the issue of strong integration between constraint 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 RA 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 NP-SQL, a non-deterministic extension of SQL with the same expressive power of *NP-Alg*. The effectiveness of *NP-Alg* and NP-SQL both as complex query and constraint modeling languages has been demonstrated by showing several queries which specify combinatorial problems.

As for future work, we plan to increase the number of polynomial cases of *NP-Alg*, in particular considering classical results on the complexity of second-order logic. Moreover, we plan to extend both languages to account for optimization problems, and to make a significantly more sophisticated implementation of NP-SQL SIMULATOR by using efficient constraint propagation techniques (e.g., by translation into propositional satisfiability [4]), and making it able to recognize the polynomial cases.

Several query languages capable of capturing the complexity class NP have been shown in the literature. As an example, in [12] an extension of datalog (the well-known recursive query language) allowing negation are proved to have such a property. A different extension of datalog, without negation but with a form of non-determinism, is proposed in [3]. 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*.

Several languages for constraint programming are nowadays available. For some of them, e.g., ECL^iPS^e [5], a traditional programming language such as PROLOG is enhanced by means of specific constructs for specifying constraints, which are then solved by highly optimized algorithms. In other modeling languages such as OPL [15] and AMPL [7], the problem is specified by means of an ad-hoc syntax. Similarly to *NP-Alg* and NP-SQL they support a clear distinction between the data and the problem description level. OPL has also a constraint programming language which allows the user to express preferences on the search methods, which is missing in the current version of NP-SQL.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley Publ. Co., Reading, Massachussetts, 1995.
2. J. Beasley, M. Krishnamoorthy, Y. Sharaiha, and D. Abramson. Scheduling aircraft landings - the static case. *Transportation Science*, 34:180–197, 2000.
3. M. Cadoli and L. Palopoli. Circumscribing DATALOG: expressive power and complexity. *Theor. Comp. Sci.*, 193:215–244, 1998.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP 2001)*, volume 2028 of *LNCS*, pages 387–401. Springer-Verlag, 2001.
5. ECL^iPS^e Home page. `www-icparc.doc.ic.ac.uk/eclipse/`.
6. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
7. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming.* International Thomson Publishing, 1993.
8. 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.
9. G. Gottlob, P. Kolatis, and T. Schwentick. Existential second-order logic over graphs: Charting the tractability frontier. In *Proc. of FOCS 2000.* IEEE CS Press, 2000.
10. ILOG optimization suite — white paper. Available at `www.ilog.com`, 1998.
11. ILOG DBLink 4.1 Tutorial. Available at `www.ilog.com`, 1999.
12. P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? *J. of Computer and System Sciences*, 43:125–144, 1991.
13. C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, Reading, MA, 1994.
14. R. Ramakrishnan. *Database Management Systems.* McGraw-Hill, 1997.
15. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.