Modular Generation of Relational Query Paraphrases

MICHAEL MINOCK
Department of Computing Science, University of Umeå, C445 MIT-Huset, Umeå 90343, Sweden (E-mail: mjm@cs.umu.se)

Abstract. This article proposes a novel technique to generate natural language descriptions for a wide class of relational database queries. The approach to describing queries is phrasal and is restricted to a class of queries that return only whole schema tuples as answers. Query containment and equivalence are decidable for this class and this property is exploited in the maintenance and use of a phrasal lexicon. The query description mechanism is implemented within the Schema Tuple Query Processor (STEP) system (http://www.cs.umu.se/~mjm/step). Because the said query class is also closed over elementary set operations, it may be reasoned with in a relatively unrestricted manner. This enables a modular separation between a reasoning component and a 'tactical' realization component. To demonstrate this modularity, this fragment is shown to be adequate for several cooperative reasoning techniques. Thus the cooperative information system serves as the 'strategic' component, deciding what to say, while the generation system acts as the 'tactical' component, deciding how to say it. Naturally expressions within the said query language are the interchange language between these two components.

Key words: cooperative information systems, logical form equivalence, modularity, natural language generation, paraphrasing SQL, relational databases, tuple relational calculus

1. Introduction

There is a long standing question within natural language generation about the degree to which a strategic component, which decides 'what to say', should be isolated from a tactical component, which decides 'how to say it'. Advocates of a high degree of integration cite the myriad ways in which an interleaved strategy, with bi-directional information flow, improves the quality of generation. Advocates of a strict, modular separation, point to the fact that separating generation into two (or more) modules has desirable engineering properties. In either case, it was observed in the early 1990's that fielded NLG systems tend to have pipelined, modular architectures with the vast majority using some type of semantic network as the knowledge representation language (Reiter, 1994). The strategic decision of content
determination usually results in an utterance specification which is realized by a tactical generator using unification grammars (Kay, 1979; Penman Project, 1989; Elhadad, 1993).

The work here adopts a modular approach in the context of providing natural language generation for cooperative information systems. Cooperative information systems are dialog systems over relational (or deductive) databases that draw their main inspiration from Grice’s maxims (Grice, 1975). Such systems are based on the semantics of relational databases, and thus the deduction is typically restricted to standard first-order logic. When information must be related to the user, utterance specifications are just fragments of logic embedded within one specific communication act or another. These communication acts may be as simple as requests for query confirmation, to intensional comments about either the database’s integrity or state relative to the user’s query. In all of these cases however, content may be expressed in the form of ‘query’ expressions, thus illustrating the primacy of query paraphrasing (Codd, 1974; Lowden and de Roeck, 1986; Ljungberg, 1991). Pragmatic aspects of specific communication acts are simply templates that fuse together any number of noun phrase query paraphrases with canned text. In summary, from the perspective of natural language generation, the cooperative information system is viewed as providing the services of a strategic generator, ‘deciding what to say’, while the query paraphraser component serves the role of tactical generator, ‘deciding how to say it’.

Appelt (1987) and Shieber (1994) pointed out a rather significant challenge to building such a modular generation system. If one is to maintain a separation between the reasoning component and the tactical component, then the reasoning component must be free to manipulate logical expressions in any way that it sees fit. In turn the tactical component must then be able describe resulting logical expressions, no matter how awkward or redundant. Since one would like to communicate the meaning expressed within the logical expression, then it seems that all equivalent logical formulas should in fact have the same sets of natural language descriptions. The alternative, to map more or less complex logic to more or less complex natural language renderings runs into practical difficulties because unrestricted deduction within the reasoner is almost certain to yield great redundancy. For example the reasoning component may call for descriptions such as the “artists that painted paintings painted by artists that are male,” when the more direct, equivalent description “Male artists that have painted paintings” is not generated because it is not recognized as equivalent.

Thus it seems, as a minimum, that the tactical component be able to recognize when complex input logical expressions are equivalent to simpler forms. This is the so called problem of logical form equivalence. Slightly stronger, logical form subsumption is a determination of whether the set of objects satisfying one formula are necessarily the superset of the objects
satisfying another formula. Clearly if one can decide subsumption then one can decide equivalence. A related and even stronger notion is the question whether all equivalent logical formulas can be reduced to a single canonical form. This shall be called the reduction to canonical form problem. Naturally the capability to solve the reduction to canonical form problem implies the solution to the logical equivalence problem, but not vice versa. Finally there is the weaker notion of logical minimization in which one may transform a given logical expression into a syntactically minimal, logically equivalent form. All of these properties are further complicated when domain knowledge enriches the system to induce a coarser grained notion of equivalence.

Focusing on just the logical form equivalence problem, for general first-order logic it has been long recognized that this problem is undecidable. This is probably what compelled Shieber to refer to such problems as 'AI-complete'. Still there have been attempted 'solutions', all partial, to the problem. There are basically two approaches: (1) alter the semantics of the logical language itself; (2) restrict logics under classical semantics to some decidable subclass. The first approach is seen as unworkable if the reasoning system is to have any type of standard representation or deductive capability. Thus the treatment here takes the second approach.

As it happens, the class of schema tuple queries formulated in the language $Q$ seems to be an ideal candidate for such an approach (Minock, 2003b). Such queries are tuple relational queries (Codd, 1972) that return only whole tuples from relations of the schema, not arbitrary combinations of projected attributes. The language $Q$ is a class of schema tuple queries which is: (1) decidable for equality, containment and disjointness; (2) closed over the basic set operations; (3) composed of disjuncts which are conjectured to be able to be reduced to canonical form. These properties are exploited in the tactical generation component which produces paraphrases of schema tuple queries specified in the language $Q$. Additionally several well known cooperative techniques (Kaplan, 1982; Shum and Muntz 1987; Gal and Minker, 1988; Imielinski, 1988; Godfrey, 1994; Chu et al., 1996; Minock and Chu, 1996; Motro, 1996) may limit their deduction to expressions within $Q$. Thus as long as the tactical component fully covers expressions in $Q$ the results of these cooperative techniques may be fully described.

1.1. Organization of this article

Section 2 presents the schema tuple query language $L$ and its closure over disjunction $Q$. It is within these languages that queries and the semantics of elementary phrases are described. Section 3 presents the tactical generator which paraphrases queries in $L$ and $Q$. Its principle knowledge input,
the phrasal lexicon, along with realization algorithm are discussed. Section 3 also includes an overview and some performance results of the system STEP within which the generation approach is implemented. Section 4 presents several well known cooperative techniques in terms of reasoning which stays closed over \( Q \). Section 5 discusses this work in the context of prior work and assess the prospects of carrying this work forward.

2. A Decidable and Closed Logical Form

Let us start with several example queries that will illustrate the logical form considered here. Assume the following relational database schema:

\[
\begin{align*}
\text{Artist} & (id, \text{name}, \text{gender}, \text{country}, \text{yearOfBirth}, \text{yearOfDeath}, \text{rank}, \text{url}) \\
\text{Painting} & (id, \text{title}, \text{year}, \text{type}, \text{school}, \text{artistId}_{\text{Artist}}, \text{rank}, \text{url}) \\
\text{HasPainting} & (\text{museumId}_{\text{Museum}}, \text{paintingId}_{\text{Painting}}, \text{VST}, \text{VET}) \\
\text{Museum} & (id, \text{name}, \text{street}, \text{city}, \text{country}, \text{rank}, \text{url})
\end{align*}
\]

The semantics here are those of standard relational databases where the primary keys are underlined and the foreign keys are subscripted by the relation to which they refer. Of note the \( \text{HasPainting} \) relation is temporal where \( \text{VST} \) stands for valid start time and \( \text{VET} \) stands for valid end time. See Elmasri and Navathe (2000) for a discussion of valid time temporal databases. See Androutsopoulos (2002) for an extensive treatment of natural language interfaces over temporal databases.

The following queries are of interest:

1. “Male artists born between 1800 and 1900.”
2. “Artists that painted major portraits.”
3. “Artists that did not paint major portraits.”
4. “Museums in Vienna that currently have paintings by Klimt or Picasso.”
5. “Artists that painted paintings housed in a museum in the artist’s home country.”
6. “Museums that have shown all the Van Gogh paintings made between 1870 and 1875.”

Queries 1–5 may be described using the current approach, though query five has a reflexive reference that currently escapes the structured authoring approach. Query six may not be expressed within the language \( Q \). Let us now define this language.

2.1. The Schema Tuple Query Languages \( \mathcal{L} \) and \( \mathcal{Q} \)

Before defining the full language \( \mathcal{Q} \), it is necessary to define the basic language in which \( \mathcal{Q} \) disjuncts are expressed, \( \mathcal{L} \). Both of these languages are
schema tuple query languages, that is tuple relational calculus formulas in which projection is disallowed.

Before describing the schema tuple queries let us start with the classical tuple relational formulas. The atomic tuple formulas are:

1. Range conditions: \( P(z) \), where \( P \) is a predicate name and \( z \) is a tuple variable.
2. Simple conditions: \( X \theta c \), where \( X \) is a component reference, \( c \) is a constant and \( \theta \) is a comparison operator.
3. Join conditions: \( X \theta Y \), where both \( X \) and \( Y \) are component references and \( \theta \) is a comparison operator.
4. Set conditions: \( X \in Y \) and \( X \not\in Y \) where \( X \) is a component reference and \( Y \) is a set of constants.

All atomic formulas are tuple relational formulas and if \( F_1 \) and \( F_2 \) are tuple relational formula, where \( F_1 \) has some free variable \( z \), then \( F_1 \land F_2, F_1 \lor F_2, \neg F_1 \) and \( (\exists z)F_1 \) and \( (\forall z)F_1 \) are also tuple relational formulas.

Now of course, the tuple calculus is just a ‘syntactic sugar’ that allows variables in queries to range over entire tuples, not over the values in the \( i \)th place of a given predicate. In any case tuple calculus has a direct translation to classical first-order logic (see Ullman, 1989).

As is common, primary keys are expressed through functional dependencies and functional dependencies are represented as universally quantified tuple formulas. Specifically the functional dependency \( W \rightarrow V \) over the relation \( P \) where \( W \) is a set of \( m \) attributes and \( V \) is a set of \( n \) attributes is expressed as the universally quantified formula: 
\[
(\forall x)(\forall y)(P(x) \land P(y) \land y.w_1 = x.w_1 \land \cdots \land y.w_m = x.w_m \Rightarrow y.v_1 = x.v_1 \land \cdots \land y.v_n = x.v_n)
\]

The symbol \( \mathcal{F} \) shall denote all of the functional dependencies holding over the schema.

**DEFINITION 2.1 (Schema tuple queries).** A schema tuple query is an expression of the form \( \{x | \varphi \} \), where \( \varphi \) is a tuple relational formula over the single free tuple variable \( x \).

Now let us restrict schema tuple queries to the sub language \( L \). First let us state some preliminary notions. A basic query component built over the free tuple variable \( x \) is a formula of the form \( (\exists y_1)(\exists y_2)\ldots(\exists y_n)\psi \), where \( \psi \) is a conjunction of range, simple, set and join conditions using exactly the tuple variables \( x, y_1, \ldots, y_n \). In the case that there are no existential variables, and thus the basic query component is just \( \psi \), it is stipulated that \( \psi \) be a single simple or set condition over \( x \).

A signed query component, \( \phi \), is either a basic query component or the formula \( \neg \varphi \) where \( \varphi \) is a basic query component. Let us refer to a signed query component as a query component and, when necessary, let us use the terms positive or negative query component to indicate sign.
DEFINITION 2.2 (The language $\mathcal{L}$). The language $\mathcal{L}$ consists of all formulas of the form:

$$P(x) \land \left( \bigwedge_{i=1}^{k} \phi_i \right),$$

where $P(x)$ restricts the free tuple variable $x$ to range over $P$ and $\phi_i$ is a signed basic query component over $x$.

See Minock (2003 a, b) for a more complete discussion of $\mathcal{L}$.

The example queries from above are shown here. Each query $\{ x | \ell \}$ returns a set of tuples. For the first five queries, $\ell \in \mathcal{L}$.

1. “Male artists born between 1800 and 1900.”
   $$\{ x | Artist(x) \land x.gender = 'male' \land x.yearOfBirth \geq 1800 \land x.yearOfBirth < 1900 \}$$

2. “Artists that painted major portraits.”
   $$\{ x | Artist(x) \land \exists y(Painting(y) \land y.type = 'portrait' \land y.rank = 'major' \land y.artistId = x.id) \}$$

3. “Artists that did not paint major portraits.”
   $$\{ x | Artist(x) \land \neg(\exists y(Painting(y) \land y.type = 'portrait' \land y.rank = 'major' \land y.artistId = x.id)) \}$$

4. “Museums in Vienna that currently have paintings by Klimt or Picasso”
   $$\{ x | Museum(x) \land (\exists y_1)(\exists y_2)(\exists y_3)
   \begin{align*}
   & (HasPainting(y_1) \land Painting(y_2) \land Artist(y_3) \land \\
   & x.id = y_1.museumId \land y_1.paintingId = y_2.id \land \\
   & y_1.VST \leq now \land y_1.VET > now \land \\
   & y_2.artistId = y_3.id \land y_3.name \in \{ 'Picasso', 'Klimt' \} \land \\
   & x.city \in \{ 'Vienna' \}) \}
   \}$$

5. “Artists that painted paintings housed in a museum in the artist’s home country.”
   $$\{ x | Artist(x) \land (\exists y_1)(\exists y_2)(\exists y_3)
   (Painting(y_1) \land HasPainting(y_2) \land Museum(y_3) \land \\
   x.id = y_1.artistId \land y_1.id = y_2.paintingId \land \\
   y_2.museumId = y_3.id \land y_3.country = x.country) \}) \}
   \}$$

6. “Museums that have shown all the Van Gogh paintings made between 1870 and 1875.”
   $$\{ x | Museum(x) \land (\forall z_1)(\forall z_2)(\exists y_1)
   (Painting(z_1) \land Artist(z_2) \land z_1.artistId = z_2.id \land \\
   z_2.name = 'Van Gogh' \land z_1.year > 1870 \land z_1.year < 1875 \Rightarrow \}$$
MODULAR GENERATION OF RELATIONAL QUERY PARAPHRASES

\[ \text{HasPainting}(y_1) \land y_1.\text{museumId} = x.\text{id} \land y_1.VST < \text{now} \land y_1.\text{paintingId} = z_1.\text{id} \]  

Query six, requiring a mixed quantifier prefix, may not be expressed using \( \mathcal{L} \). Naturally all of these queries may be expressed using standard SQL.

The closure of \( \mathcal{L} \) over disjunction leads to the language \( \mathcal{Q} \).

DEFINITION 2.3 (The language \( \mathcal{Q} \)). \( q \in \mathcal{Q} \) if \( q \) is written as a finite expression \( \ell_1 \lor \cdots \lor \ell_k \) where each \( \ell \in \mathcal{L} \) and \( \ell \) is over the free tuple variable \( x \).

2.2. Reasoning over \( \mathcal{Q} \)

One may determine if a given query in \( \mathcal{Q} \) (or \( \mathcal{L} \)) is satisfiable.

THEOREM 2.4 (\( \mathcal{Q} \) is decidable for satisfiability) (Minock, 2003a). For all \( q \in \mathcal{Q} \) over the free variable \( x \), one may determine if there exists a database instance \( d \) respecting the functional dependencies \( \mathcal{F} \), where for some schema tuple \( \tau, \{\tau/x\}q \).

One may decide subsumption, equivalence and disjointness between expressions written in \( \mathcal{Q} \) and, thus, by definition \( \mathcal{L} \). As will be made clear in the next section, the ability to calculate subsumption is very important in the query description process.

THEOREM 2.5 (Decidability of \( \subseteq,= \) and disjointness over \( \mathcal{Q} \)) (Minock, 2003a). if \( q_1 \in \mathcal{Q}, q_2 \in \mathcal{Q} \) and \( \mathcal{F} \) expresses the functional dependency constraints over the domain, then there exists a sound and complete inference mechanism to decide if the three predicates:

1. \( \{x_1|q_1\} \subseteq \{x_2|q_2\} \)
2. \( \{x_1|q_1\} = \{x_2|q_2\} \)
3. \( \{x_1|q_1\} \cap \{x_2|q_2\} = \emptyset \).

are necessarily true over the set of all database instances for which \( \mathcal{F} \) holds.

The following theorem is a natural consequence of DeMorgan’s rule in the schema tuple query case.

THEOREM 2.6 (Syntactic query difference is closed over \( \mathcal{Q} \)) (Minock, 2003). Let \( q_1 \in \mathcal{Q} \) and \( q_2 \in \mathcal{Q} \). Then there is a \( q_3 \in \mathcal{Q} \) such that for all database instances \( \{x_1|q_1\} - \{x_2|q_2\} = \{x_1|q_3\} \).
The consequences of this theorem are that one may: (1) complement queries; (2) compute the difference of two queries as a third query expression; (3) syntactically calculate query intersections. In fact all the classical set operations are closed over $Q$. Thus one may employ unrestricted set based reasoning over queries in $Q$ and may report results in natural language if one has a tactical component that describes all queries within $Q$.

2.3. Template queries

A template query is a schema tuple query defined using a formula in which constants may be parameters. Thus the query $\{x | Artist(x) \land x.gender = c\}$ is a template query where the parameter $c$ could be the constant ‘male’ or ‘female’. One may also have set valued parameters as in: $\{x | Painting(x) \land x.type \in c\}$. As a notational convention, let us represent a template query, $\{x | q_1\}$ as $\{x | q_1^{C_1}\}$ where $C_1$ is the set of all the parameters within the query.

A difficulty occurs when one is to consider if one template query subsumes another. The semantics here is to assume that a template query represents the set of all possible answer tuples, when parameters from the template query are bound to all combinations of constants drawn from some large finite set $\mathcal{C} \subseteq \mathcal{U}$. Thus the template query $\{x | q_1^{C_1}\}$ subsumes $\{x | q_2^{C_2}\}$ if and only if for all bindings of $C_2$ over $\mathcal{C}$ there exists a binding for $C_1$ over $\mathcal{C}$ such that $q_1$ under the bindings of $C_1$ subsumes $q_2$ under the bindings of $C_2$. There are no problems deciding such a subsumption relationship, because $\mathcal{C}$ is finite and Theorem 2.5 holds. Under this semantics the following subsumption relationships hold:

$$\{x | Artist(x)\} \supseteq \{x | Artist(x) \land x.gender = c\}$$
$$\{x | Artist(x) \land x.gender = c\} \supseteq \{x | Artist(x) \land x.gender = \text{‘male’}\}$$
$$\{x | Artist(x) \land x.gender = c\} \supseteq \{x | Artist(x) \land x.gender = c’\}$$
$$\{x | Artist(x) \land x.gender = c\} \supseteq \{x | Artist(x) \land x.gender = c’ \land x.country = c”\}$$

The following subsumption relationships do not hold:

$$\{x | Artist(x) \land x.gender = c\} \nsubseteq \{x | Artist(x)\}$$
$$\{x | Artist(x) \land x.gender = \text{‘male’}\} \nsubseteq \{x | Artist(x) \land x.gender = c\}$$
$$\{x | Artist(x) \land x.gender = c \land x.country = c’\} \nsubseteq \{x | Artist(x) \land x.gender = c”\}$$

A curious question surrounding this process, is whether one should enforce the binding of parameters to be consistent with the attribute domains over which they range. Currently this constraint is not enforced, and thus the following subsumption relationship does not hold:

$$\{x | Artist(x) \land x.gender = c\} \nsubseteq \{x | Artist(x) \land x.gender \neq c’\}$$

From a logical point of view the definition of template query subsumption could be strengthened along the lines discussed in Deemter (1998). The current virtue of the weak approach is that it is easier to implement and it results in finer grained subsumption hierarchies (see ahead to 3.1.5).
2.4. Reduction to canonical form

The question of whether one may reduce $\mathcal{L}$ expressions to canonical form remains open. More progress has been made on the related, and perhaps more practical question of reducing $\mathcal{L}$ (and $\mathcal{Q}$) expressions to a minimal form. Given a expression $l \in \mathcal{L}$ let us say that $l'$ is an equivalent rewriting of $l$ if there is a finite sequence of semantically equivalent formulas $l_1, \ldots, l_m$ such that $l_1$ is $l$ and $l_m$ is $l'$ and each $l_{i+1}$ is obtained from $l_i$ by applying a rewrite rule. The set of rewrite rules are too extensive to document here, but they include the routine rewrite rules (Ullman, 1989) and several special cases designed around the syntactic form of $\mathcal{L}$. Of interest there are additional rewrite rules that test if two distinct existential variables may be merged or if one may drop variables, components or conditions wholesale. In such cases, an explicit equivalence test is issued to insure that a semantically equivalent query has been written. In summary, the approach here employs a set of sound, but not necessarily complete set of rewrite rules for queries in $\mathcal{L}$. These rewrite rules are used to search for shorter (in terms of the number of conditions) equivalent rewritings of a query. In practice one is able to simplify a wide class of queries within reasonable time.

The reason that such a capability is important relates to the need to systematically gather syntactic material for recursive calls to the generation paraphrase mechanism. Such recursive calls are made when the paraphraser shifts focus to another tuple variable (see Section 3.2.2).

3. The Tactical Generation Component

The tactical generator here consists of a core generation component seated within a full generation component. The core component describes expressions within $\mathcal{L}$, while the full component processes application dependent communication act commands with multiple $\mathcal{Q}$ or $\mathcal{L}$ arguments. The output from the core component is a highly aggregated noun phrase. The output of the full component is a sentence or a paragraph of canned text wrapped around results obtained from the core component.

Since the author of the generation system is assumed to be a database administrator, not a linguist, the approach here is phrasal. Moreover the administrator is given a structured method of authoring a phrasal lexicon so that they may declare the schema covered with respect to a given class of queries. For the core query description generator this well defined class currently corresponds to the non-cyclic $\mathcal{L}$ queries using only equality-joins over conceptual models limited to only binary relationships. This said, the administrator may always add additional ad hoc entries to extend the coverage to $\mathcal{L}$ queries beyond this class.
3.1. Knowledge inputs – the phrasal lexicon

Along with the schema, the phrasal lexicon is the only knowledge input to the core tactical generator. The phrasal lexicon consists of a set of \( n \) entries each grouping a single elementary template query with a sequence of patterns. The template query is limited to the language \( L \) and the patterns each express linguistically the meaning of the template query. The \( i \)th entry of the phrasal lexicon, \( e_i \) is \( \langle \{ x | \ell_i \} : p_{i,1}, \ldots, p_{i,m_i} \rangle \) where \( \{ x | \ell_i \} \) is a template query and \( p_{i,1}, \ldots, p_{i,m_i} \) are \( m_i \) patterns.

A pattern is simply a phrase that consists of three constituents: "[modifier] head [complement]". Constituents consist of plain text, possibly including template query parameters or calls to simple morphological functions over parameters. As discussed below, constituents may also contain recursive calls to describe a sub-query. The modifier, head and complement distinction is best illustrated with an example. The simple description, "male artists from France" has "male" as a modifier, "artists" as the head, and "from France" as the complement. Thus it would be represented: "[male] artists [from France]". The constituent syntax for modifier and complement constituents denotes that constituents may collect in such positions during aggregation. By contrast there can only be one head constituent. Thus one may generate the aggregated pattern "[major, male] artists [from France, that have paintings in the Louvre]".

Let us now cover the different types of entries within the phrasal lexicon.

3.1.1. Simple Entries

Let us start with the simplest type of entry. Here is the entry for the condition free database relation Painting:

\[ e_1 : \langle \{ x | \text{Painting}(x) \} : \text{"[ ] paintings [ ]"} \rangle \]

Now here is the entry for the relation Painting, attribute type and operator =:

\[ e_2 : \langle \{ x | \text{Painting}(x) \land x.\text{type} = c \} : \\
\text{"[ ] paintings [of the type c]"} \\
\text{"[c] paintings [ ]"} \rangle \]

For the first pattern the lead '[ ]' specifies that there is an empty modifier while the head is "paintings" and the complement is "[of type c]". The complement has the parameter \( c \) from the template query. The second pattern describes an alternate way of expressing the template query.

Naturally one may also have constants specified in the queries as well.

\[ e_3 : \langle \{ x | \text{Painting}(x) \land x.\text{type} = \text{‘portrait’} \} : \\
\text{"[ ] portraits [ ]"} \rangle \]
Strictly speaking, entries with such domain specific constants belong to the class of extended entries. They merely help to sharpen up the generated language but are not absolutely required for coverage. Also note that entry $e_3$ specifies a new head that, as we shall see, overrides the head specified in $e_1$.

Although the generation process will be covered in detail below, it should be intuitive that given the three entries above one may generate the description of the query $\{x | \text{Painting}(x) \land x.\text{type} = \text{‘portrait’}\}$ as “paintings of the type portrait”, “portrait paintings”, or “portraits”. The simple heuristic employed here prefers the last form over the first two.

To declare the phrasal lexicon complete with respect to non-join queries, one must cover all of the attribute/operator combinations for each relation in the schema. If there are 6 comparison operators ($>, \geq, =, \neq, <, \leq$) and two set operators ($\in, /\in$), and each attribute/operator combination makes sense, then the minimum number of simple entries required by the schema in Section 2 is: $(8 \times 8 + 1) + (8 \times 8 + 1) + (8 \times 4 + 1) + (8 \times 7 + 1) = 220$. The ‘+1’ terms cover the entry for the case where no conditions are applied. Of these 220 entries, only a portion of them are expected to be used in practice. Assuming that only numeric attributes are involved in non-equality comparison conditions, only no-order values are in set conditions and ID and URL attributes are not involved in simple or set conditions, this reduces to $(4 \times 4 + 6 \times 2 + 1) + (4 \times 4 + 6 \times 1 + 1) + (6 \times 2 + 1) + (4 \times 5 + 1) = 86$ simple entries to cover all non-join queries over the schema of Section 2.

3.1.2. Join entries: One-to-many and Many-to-one Relationships

Now let us face the issue of representing the entries associated with shifting focus from a given tuple variable to another through traversing a join. First is the simple case of joining over foreign keys. Such joins account for the bulk of meaningful many-to-one and one-to-many relationships. In Section 3.1.3 the more complex case involving many-to-many relationships shall be considered.

Each attribute that can be meaningfully joined with another must be considered. Assuming that only equality joins are to be considered, then for the example of Section 2 this would amount to a total of 4 attribute matches: $\text{Artist.id} \leftrightarrow \text{Painting.artistId}$, $\text{Painting.id} \leftrightarrow \text{HasPainting.paintingId}$, $\text{HasPainting.museumId} \leftrightarrow \text{Museum.id}$ and $\text{Museum.country} \leftrightarrow \text{Artist.country}$. The joins with $\text{HasPainting}$ may be side-stepped for now because they are covered in the many-to-many case below. Additionally the join $\text{Museum.country} \leftrightarrow \text{Artist.country}$ is excluded because of its semantic irrelevance. This leaves us with only $2 \times 2 \times 1 = 4$ entries to cover this space of equality joins. The reason for the first doubling is that one must take into account direction when one describes joins. The second doubling occurs because one must also
consider the negative case in which the query specifies that answers do not participate in such relationships. Sign and directionality of the join shall be indicated by $R_1.a_1 \rightarrow^+ R_2.a_2$.

For the join involving $Painting.artistId \rightarrow^+ Artist.id$ there is the entry:

$$e_4 : \{x | Painting(x) \land \neg(\exists y)(Artist(y) \land x.artistId = y.id \land \psi)\} : \text{"[ ] paintings [by $GEN(\{y | Artist(y) \land \psi\})$"]}$$

The key issue to note here is that the complement phrase has a recursive call to the generator ($GEN(\ell)$). This will cause a completely new generation problem to be instantiated. Additionally this entry leaves as unknown the formula $\psi$ which, as we shall see below must be derived during generation. The negative case is almost identical to the entry above. The entry for $Painting.artistId \rightarrow^− Artist.id$ is:

$$e_5 : \{x | Painting(x) \land \neg(\exists y)(Artist(y) \land x.artistId = y.id \land \psi)\} : \text{"[ ] paintings [not by $GEN(\{y | Artist(y) \land \psi\})$"]}$$

Assuming that there are simple entries covering artist, one may now generate descriptions such as “[ ] [paintings] [not by ([male] artists [from France])]”.

As a final note, because entry 5 is a simple alteration of its corresponding positive entry 4, it seems reasonable to assume that negative join entries may be automatically derived from their corresponding positive entries. Thus only $2 \times 1 = 2$ entries are required to cover the space of one-to-many equality joins for the schema of Section 2.

3.1.3. Join entries: Many-to-many Relationships

Now let us consider the case of many-to-many type relationships. These present a special, though not insurmountable difficulty. There is only one many-to-many relationship in the schema of Section 2. This is the relationship between $Painting$ and $Museum$ through the relation $HasPainting$. Somewhat vexing is the fact that the temporal attributes, VST (valid start time) and VET (valid end time) are involved within the many-to-many relationship. Consider the entry:

$$e_6 : \{x | Paintings(x) \land$$
$$\quad (\exists y_1)(\exists y_2)(HasPainting(y_1) \land Museum(y_2) \land$$
$$\quad x.id = y_1.paintingId \land y_1.museumId = y_2.id \land$$
$$\quad y_1.VST \leq \text{now} \land y_1.VET > \text{now} \land \psi)\} :$$
$$\text{"[ ] paintings [currently in $GEN(\{y_2 | Museum(y_2) \land \psi\})$"]}$$

The only way to precisely control this is to make multiple join entries for each combination of given attributes in the joining relation. Though many of the combinations are meaningless a possible bound on the
coverage of positive many-to-many relationships where $a$ is the number of non-join attributes in the relationship, is $2 \times 2 \times (8 + 1)^a$. Thankfully relations that bridge many-to-many relationships are often of few attributes. If we assume negative cases may be derived from positive cases and that numerical attributes are not involved in set conditions, then we would need $2 \times (6 + 1)^2 = 98$ entries to fully cover the many-to-many relationship between museums and paintings, though this is an exaggerated bound.

3.1.4. Extended Entries

If one supplies the full set of 186 entries along the lines detailed above, one may declare the schema to be covered by the phrasal lexicon. That is the tactical component may generate descriptions of all non-cyclic $\mathcal{L}$ queries, limited to only equality-joins, over the schema of Section 2.

Naturally one may extend the phrasal lexicon by extending it to cover more specific language. For example, the following entry would simplify some descriptions:

$$e_7: \langle \{ x | \text{Artist}(x) \land x.\text{country} \in \{ \text{Sweden}, \text{Denmark}, \text{Norway} \} \rangle: \text{"[Scandinavian] artists [ ]"}$$

3.1.5. The Subsumption Hierarchy

At system start up, all the entries of the phrasal lexicon are automatically compiled into a subsumption hierarchy that records the semantic relationships between entries. The notion of subsumption hierarchy here corresponds with that used in description logics, however it should be noted that the logic here differs considerably from the unary and binary relations of typical description logics. See Minock (2003b) for a full discussion of this. In any case the phrasal lexicon is compiled into a subsumption hierarchy by sorting entries down into the hierarchy. Figure 1 shows the compilation of a set of entries over the schema of Section 2. Note that this phrasal lexicon gives reasonable, but not complete coverage over the schema.13

3.2. The Core Generation Process

In summary the core query paraphrase process works by (1) semantically sorting an input $\mathcal{L}$ query into the subsumption hierarchy; (2) combining the patterns of the sorted query's immediate parents into a phrasal description of the query. Let us begin with a very simple example, followed by a more rigorous description of these two phases. In addition to the simple entries documented in Section 3.1.1, let us assume the phrasal lexicon also contains the following entry:

$$e_8: \langle \{ x | \text{Painting}(x) \land x.\text{school} \in c \} : \text{"[ ] paintings [of the c school]"}$$
Now suppose that the query for the “landscape paintings of the impressionist or cubist school” needs to be described. The input query is, \( \{ x | \text{Painting}(x) \land x\.type = \text{‘landscape’} \land x\.school \in \{ \text{‘cubist’, ‘impressionist’} \} \} \).

When this query is sorted into the subsumption hierarchy, entries \( e_2 \) and \( e_8 \) are identified as being the query’s immediate parents. Now at a logical level the input query may be rewritten as a combination of the filled in template queries of these two entries. That is \( \{ x | \text{Painting}(x) \land x\.type = \text{‘landscape’} \land x\.school \in \{ \text{‘cubist’, ‘impressionist’} \} \} = \{ x | \text{Painting}(x) \land x\.type = \text{‘landscape’} \} \cap \{ x | \text{Painting}(x) \land x\.school \in \{ \text{‘cubist’, ‘impressionist’} \} \} \). This illustrates that in this case the immediate parents form the basis of an equivalent rewriting of the input query. All that is required is to ground the template queries of entries \( e_2 \) and \( e_8 \) with the constants from the input query.

As long as the patterns associated with each grounded entry may be coherently combined, we may obtain an exact description of the input query. The patterns associated with grounded entries \( e_2 \) and \( e_8 \) may be conjoined through pooling modifiers and complements around a single head. Excluding the permutations of the ‘cubist’ and ‘impressionist’, the possible ways to combine the 2 patterns of \( e_2 \) and the 1 pattern of \( e_8 \) are:

1. \([\text{landscape}] \text{ paintings [of the cubist or impressionist school]}\)
2. \([\ ] \text{ paintings [of the type landscape, of the cubist or impressionist school]}\)
3. \([\ ] \text{ paintings [of the cubist or impressionist school, of the type landscape]}\)
Using a simple heuristic of minimizing sentence length, the first choice is preferred. If one extends the heuristic to communicate the maximum amount of information in the first \( k \) symbols, the exact ordering 1–3 above is induced.

Let us now delve into the finer details of the paraphrase process, including the issue of shifting focus through join entries.

### 3.2.1. Sorting the Input Query into the Hierarchy

As illustrated in the example above, through sorting a non-join query into a complete subsumption hierarchy, one may obtain an equivalent rewriting of the query. This equivalent rewriting is obtained through conjoining the grounded template queries of the sorted query’s immediate parents. Because the complete hierarchy contains entries for all elementary conditions, it should not be difficult to see that an equivalent rewriting may always be obtained for non-join input queries.

A more complex case arises when the input query involves joins. In such a case the immediate parents may be join entries in which an open formula \( \psi \) must be solved for with respect to the input query. As an example assume that the input query is \( \{ x | \text{Painting}(x) \land x.\text{type} = \text{‘portrait’} \land (\exists y)(\text{Artist}(y) \land y.\text{country} \in \{ \text{‘Sweden’}, \text{‘Denmark’}, \text{‘Norway’} \} \land x.\text{artistId} = y.\text{id}) \} \). In this case the simple entry \( e_3 \) and the join entry \( e_4 \) are the immediate parents of the input query. The open formula \( \psi \) of \( e_4 \) is \( y.\text{country} \in \{ \text{‘Sweden’}, \text{‘Denmark’}, \text{‘Norway’} \} \). Though this example is small, the general technique is to reduce the input query to canonical form so that one may ‘walk down’ its structure and piece together the material for \( \psi \). Once the set of sub-formulas \( \psi_1, \ldots, \psi_m \) are obtained for each join entry parent, these formulas are substituted into their corresponding join entries and the grounded parent entries indeed form an equivalent rewriting of the input query. Of course the linguistic patterns associated with join entry parents must be solved for through recursive calls to the entire generation process. In the example here the call \( \text{GEN}(\{ y | \text{Artist}(y) \land y.\text{country} \in \{ \text{‘Sweden’}, \text{‘Denmark’}, \text{‘Norway’} \}) \) is issued.

While this process of solving for open formula is relatively straightforward for the case of non-cyclic queries over schemas with only binary many-to-many relationships, a general solution to the problem is wrapped up in the full canonicalization question of Section 2.4. As an example of the need for a general solution, consider the handling of superlatives such as “Give the oldest German artist.”. This may be expressed in \( \mathcal{L} \) as \( \{ x | \text{Artist}(x) \land x.\text{country} = \text{‘Germany’} \land \neg(\exists y)(\text{Artist}(y) \land y.\text{country} = \text{‘Germany’} \land y.\text{yearOfBirth} < x.\text{yearOfBirth}) \} \). While this exact case may be handled in an ad hoc manner, a general expression for the ‘oldest’ artist with some property would be \( \{ x | \text{Artist}(x) \land \psi \land \neg(\exists y)(\text{Artist}(y) \land \ldots) \} \).
\( \psi' \land y.\text{yearOfBirth} < x.\text{yearOfBirth} \) \) where \( \psi \) expresses the property for the variable \( x \) and \( \psi' \) expresses the same property for the variable \( y \). Further research will attempt to extend the techniques here to handle such cases.

3.2.2. Combining Pattern Sets to Generate Plain Text Descriptions

Given that a set of grounded parent entries, extended with sub-formulas, form an exact re-writing of the query, one may select combinations of patterns, one from each entry, to generate a phrasal description of the query. Phrasal descriptions are simple aggregations of sets of phrases, where all the modifier and complement constituents are pooled into sets around a single head.

There are two issues that complicate this process. The first issue concerns heads that override the default value for the relation. The entry \( e_3 \) does this for the relation \( \text{Painting} \) which has its default head specified in the condition free entry \( e_1 \). The algorithm stipulates that at most one pattern of a combination may override the default head. The second complicating factor is that occasionally the whole generation process recurs when there is a \( \text{GEN} \) call over a sub-formula within a constituent of a pattern. In this case all the resulting textual material is placed within the constituent that issues the recursive call.

3.3. The Full Tactical Generator

Thus far we have only considered generating paraphrases of queries within \( \mathcal{L} \). There are two principle extensions to be considered: (1) generating descriptions of query expressions in \( \mathcal{Q} \); (2) pragmatic aspects of processing communication act commands from the reasoning component.

3.3.1. Generating Descriptions of \( \mathcal{Q} \) Query Expressions

The obvious approach to generating descriptions of \( \mathcal{Q} \) queries is to generate a description of each disjunct of the query (in \( \mathcal{L} \)) and to coordinate the resulting noun phrases with ‘and’ or ‘or’ depending on the context. Thus the description of the query \( \{x|(\text{Artist}(x) \land x.\text{country} = \text{‘France’}) \land x.\text{gender} = \text{‘female’}) \lor (\text{Artist}(x) \land x.\text{country} = \text{‘Germany’}) \lor (\text{Artist}(x) \land x.\text{country} = \text{‘Germany’} \land x.\text{gender} = \text{‘male’})\} \) might be “female artists from France, artists from Germany and male artists from Germany”.

Certainly, as the example points out, intensional descriptions within \( \mathcal{Q} \) should be simplified before they are presented. This would give us the more compact description “artists from Germany and female artists from
France.” Currently the only simplification that is performed is to remove any disjunct that is subsumed by another disjunct.

More generally simplification may be envisioned as a search process where utility is inversely correlated with the number of \( L \) disjuncts in the \( Q \) expression. The search operators are derived from the basic set operations and the notion that subsumed disjuncts may be removed. Given this, it is likely that some type of hill-climbing or annealing search will operate the best. It should be noted however, that there is not a unique canonical form for expressions in \( Q \) thus, in general, one will not know whether they have found the best simplification.

Another, more extended idea is to conduct the minimization over equivalent algebraic expressions of \( L \) terms involved in set operations such as intersection, complement and difference. In such a case one may generate a description such as “artists in France or Germany that are not male artists from France”.

### 3.3.2. Realizing Specific Communication Acts

The actual requests from the reasoning component will involve saying things about the queries being described. For example it may be that the cooperative information system wishes to inform us of the fact that in the current database state there are no tuples that satisfy the query \( q_1 \). In such a case a corresponding communication act command must be supported, for example \( \text{InformNoTuples}(q) \). This command, \( \text{InformNoTuples}(q) \), is basically the canned text template “There are no GEN(\( q \)).” Needless to say, hard-coded commands need to be written for all the supported communication acts.

### 3.4. The System Step

The generation approach described above is implemented within the system Schema Tuple Expression Processor (STEP). STEP is a general purpose system for reasoning over schema tuple queries specified using \( Q \). The core of STEP performs basic reasoning services over schema tuple query expressions. STEP translates queries specified in \( L \) into expressions in domain calculus and then uses the theorem proving system SPASS (http://spass.mpi-sb.mpg.de/) to obtain satisfiability decisions. The basic reasoning services are subsumption tests and syntactic query differencing and intersection.

STEP maintains a phrasal lexicon, and, at start up time, STEP compiles this phrasal lexicon into a subsumption hierarchy. When queries need description they are sorted into this hierarchy, direct parent patterns are obtained, and patterns are combined to generate a single sentence natural language descriptions. On a DELL Dimension 8300 running Debian Linux,
CLISP and SPASS 2.1, it took 10.97 seconds of CPU time (27.4 s real time) to compile the phrasal hierarchy in Figure 1. This task requires 901 satisfiability tests to be issued to SPASS. The times to generate the descriptions for queries 1 to 4 took 0.45 CPU (1.10 real), 0.40 CPU (1.10 real), 0.31 CPU (0.78 real), 0.43 CPU (1.08 real) seconds respectively. Thus it must be said, *STEP is sufficiently efficient to generate real time query descriptions.*

4. Cooperative Reasoning over $Q$

This section shows that several cooperative techniques$^{14}$ may be implemented by reasoners that restrict their representation language to $Q$. Again the idea is to let the cooperative information system determine relevant content, and then to depend on the tactical component to express such content. We now discuss specific cooperative techniques.

4.1. Query misconceptions

A user may have a *query misconception* (Gal and Minker, 1988), meaning that the user is unaware that their query presupposes an illegal state of the database. A system is more cooperative when it identifies and reports a query misconception rather than just responding “no answers”. In a large number of cases a misconception is identified through a simple satisfiability check of the query. Thus the query $\{x | \text{Artist}(x) \land x.\text{gender} = 'male' \land x.\text{gender} = 'female' \}$ is immediately flagged as a misconception. For queries in $Q$ this may be decided in general based on Theorem 2.4.

Simply reporting that a query is a misconception is often not as useful as reporting which part of the query caused the misconception. To achieve this the unsatisfiable query $l$ must be partitioned into two queries $l_1$ and $l_2$ (i.e. $l = l_1 \land l_2$) where both $l_1$ and $l_2$ are satisfiable. Thus in the query above $l_1$ is $\{x | \text{Artist}(x) \land x.\text{gender} = 'male' \}$ and $l_2$ is $\{x | \text{Artist}(x) \land x.\text{gender} = 'female' \}$. This would result in the report: “It is impossible that $\text{GEN}(l_1)$ are also $\text{GEN}(l_2)$”.

To cover the case that there are irrelevant conditions, such as in the query $\{x | \text{Artist}(x) \land x.\text{gender} = 'male' \land x.\text{gender} = 'female' \land x.\text{country} = \text{'Germany'} \}$, the approach is strengthened by stipulating that $l_1$ and $l_2$ are the minimal, satisfiable sub-queries of $l$ where $l_1 \land l_2$ is unsatisfiable. Though the calculation of such partitions are likely to result in an exponential number of satisfiability tests, the reader is reminded that such complexity is in terms of query lengths.

Though simple satisfiability tests followed by a query partition flags many query misconceptions, there are cases where specific domain knowledge is the root of a misconception. For example assume that a user asks for “all the 20th century German artists that painted still lives dated
between 1850 and 1890.” This is clearly a misconception of the database schema. The user must be informed that a painting necessarily has a year that is after the year of birth of the painter. Such a conceptual response is in the form of a ‘query’.

To see how this is achieved, assume that illegal states of the database are represented by the expression $\beta \in Q$. In the example above one disjunct of $\beta$ is $\text{Artist}(x) \land (\exists y_1)(\text{Painting}(y_1) \land y_1.\text{artistId} = x.id \land x.\text{yearOfBirth} > y_1.\text{year})$. As a routine part of processing, disjuncts from $\beta$ are syntactically subtracted from the user’s query. Theorem 2.6 guarantees the correctness of such a process. Needless to say if the user’s query is canceled out by such a process, then the disjunct from $\beta$ that caused this cancellation should be reported back to the user as a misconception in their query.

### 4.2. False presuppositions

Related to query misconceptions, a user should be made aware of any false presupposition they have about the database state. A false presupposition is an assumption that is implicit in a user's query, though false. The difference between a false presupposition and a misconception is that a false presupposition is an assumption that happens, in the current case, not to hold. In contrast a misconception never holds over in a legal state of the database.

The system CO-OP (Kaplan, 1982) used a limited theory of cooperation to correct false presuppositions. For example assume that a user requests, “give the museums in Bratislava that have portraits painted by Goya from year 1767 to 1776.” Suppose that there are no such paintings. A traditional system would stone wall the user with the answer “none.” A system that could detect false presuppositions would perhaps respond, “there are no paintings by Goya in museums within Bratislava.” This is a description of the minimal failing sub-query (MFS) of the users original query. Efficient algorithms exist to find MFSs as well maximal succeeding sub-queries (MSS) (Godfrey, 1994). The CARMIN (Godfrey et al., 1994) system implements this work and includes an integrated explanation and answer presentation system based on the proof path used by a PROLOG meta-interpreter. Aspects of what to include and how to coordinate these explanations are also addressed (Gaasterland and Minker, 1991).

In practice a large number of false presuppositions are simply the result of not using the correct domain values for attributes. For example the query $\{x | \text{Artist}(x) \land x.\text{country} = \text{‘Grance’} \land x.\text{yearOfBirth} > 1900\}$ contains a false presupposition that there is a country named ‘Grance’. The MFS, $\{x | \text{Artist}(x) \land x.\text{country} = \text{‘Grance’}\}$, should be described back to the user. Namely “there are no artists from Grance”.


dedicated to generating relational query paraphrases.
For queries in $\mathcal{L}$ the strategy to identify MFSs is to start with simple tests on attribute values looking for an empty answer set. Thereafter the search is broadened to include multiple conditions over single tuple variables. If failing queries have still not been identified, then the search is continued over join paths incorporating larger and large numbers of connected tuple variables.

4.3. Query relaxation

Query relaxation is useful when a query has no matching tuples. During query relaxation conditions may be weakened or alternate entity types may be queried. For example when asking for “the 18th century Swedish artists with paintings in the Prado”, one may relax the search to museums in the vicinity of Madrid, to Scandinavian artists or perhaps to Swedish artists of the 17th or 19th centuries. In any case it is important to describe the relaxed query to the user before flooding the user with extensional answers. Natural language descriptions of the user query and relaxation process are generated for the cooperative information system CoBase (Chu et al., 1996; Minock and Chu, 1996).

Similar to the Type Abstract Hierarchies (TAH) structure in CoBase (Chu et al., 1996), the approach here is to induce a conceptual hierarchy over the given database off-line. In the case here this is simply a subsumption hierarchy of ground queries, termed the ground query hierarchy. The process of query relaxation or, in the next subsection, of query specialization may be seen merely as sorting a given query into the ground query hierarchy and then ‘walking the hierarchy’ to transform the query. If queries are limited to $\mathcal{L}$ this process is relatively straightforward. Thus the query seeking the 18th century Swedish artists with paintings at the Prado, is sorted under: (1) Scandinavian artists; (2) artists with paintings in museums in Madrid; (3) 17th century artists; (4) 19th century artists. The initial query, as well as these immediate generalizations, may all be expressed in $\mathcal{L}$ and when the input query $l_1$ is relaxed to the answer generating $l_2$, the user may receive the report, “There are no $GEN(l_1)$, but there are $GEN(l_2)$”.

4.4. Intensional query answering

Intensional query answering (Shum and Muntz, 1987; Imielinski, 1998) provides a summary answer rather than the entire tuple extension satisfying the query. If you are asking for all “the paintings of Van Gogh in the Hague,” it is perhaps better to receive the answer “all of Van Gogh’s still lives painted between 1878 and 1888”. Rather than the literal set of paintings. This intensional response is in fact more informative in the case that the user does not interpret an enumeration to in fact be “all the paintings to be all of Van Gogh’s still lives made between 1878 and 1888”. Naturally such a summary may be specified as a query expression.
The approach here is to adopt the protocol that if the number of answers to a user’s query exceed a set amount, then the user is given the option of specializing their query to direct children in the ground query hierarchy described in Section 4.3. So if the user asked for all the artists with paintings in Madrid, they might receive the option specializing their query to the 18th century artists at the Prado, the 18th century artists at the Reina Sofia, etc.

4.5. Completeness/incompleteness reports

Often one is prepared to make very specific declarations of completeness over their databases. For example one might say that their database has “all the major artist of the 19th century and all the major paintings currently in the Louvre”. When a user queries such a database they may wish to know what portion of their query may be answered completely. Thus if the users ask for the “major 19th and 20th century German artists”, they should be informed that their answer is complete for the major 19th century artists, but perhaps only partial for the major 20th century German artists.

More abstractly, consider that there are a set of $n$ views: $v_1, \ldots, v_n$ that describe the portion of the schema for which the database has complete coverage. The problem is, given a query $q$, calculate the portion of $q$ that may be answered with certainty using the views. The answer to this problem is a disjoint partition of the original query $q$ into two queries $q_{\text{certain}}$ and $q_{\text{uncertain}}$ where $q_{\text{certain}}$ is the portion of the query that may be answered with certainty and $q_{\text{uncertain}}$ is the remainder that may not be answered with certainty over the views.

In Minock (2004) it is shown that if sound and complete materialized views are described using the language $\mathcal{L}$ without negative components, and the user’s query is specified using the language $\mathcal{L}$, then one may generate intensional descriptions (within the language $\mathcal{Q}$) of the certain answers of the query over the views. Moreover one may directly apply the resulting query over the materialized views to obtain such answers. Finally one may obtain an intensional description of the portion of the query that may not be answered with certainty over the materialized views.

It should be mentioned that this problem of completeness reports was addressed by the PANORAMA system (Motro, 1996). PANORAMA allows users to attach properties to views and then decides which other views inherit such a property. PANORAMA’s view definition language is strictly conjunctive and the set of properties include soundness, completeness and emptiness, but it is an open, extensible set. PANORAMA associates a meta-schema with the actual schema where the meta-schema encodes the view definitions with associated properties. Queries are processed over both the base relations as well as the meta relations. The
normal extensional answer is augmented with a set of view/property pairs that cover the extensional answers. The technique is efficient, yet incomplete. That is not all relevant covering views will be reported in the meta-answer.

Motro terms his approach algebraic and contrasts it with the logical based work in semantic query optimization (Chakravarthy et al., 1990).

4.6. INTEGRATED COOPERATIVE INFORMATION SYSTEMS

This paper proposes a modular approach in which the cooperative information system decides ‘what to say’ and the generation system description decides ‘how to say it’. The cooperative information system consists of sub-systems that perform misconception detection, false presupposition detection, query relaxation, intensional answer generation and completeness reasoning.

Figure 2 shows the flow of control of the cooperative information system. The user is assumed to compose their logical query through some type of query formulator. This may be as primitive as a text box in which to type a logical query expression (e.g. SQL), to simple forms, to point and click query constructors (Thompson et al., 1983), to a full natural language understanding system. In any case, after construction, misconception detection is followed by query confirmation. Given the query is confirmed, either the
query will need to be generalized if it is not returning answers, or specialized if it is too broad. The generalization step also performs false presupposition detection as one of its internal checks. Finally, once an adequate answer generating query is obtained, the system will generate a completeness report of the query with respect to the available data before the actual answers are presented. Note that each of these steps will generate a stream of communication act commands will be solved by the tactical component. There will be simple mechanisms which enable the dynamic suppression of commentary as well as methods by which the user may interrupt the system and manually control the actual cooperative processes.

5. Discussion

The language \( Q \) (and its base \( L \)) play a central role in the work here. \( Q \) is essentially the schema tuple queries limited to the \( \exists^* \forall^* \) quantifiers closed over disjunction. Though this is within the Schönfinkle-Bernays class, a well known decidable fragment of first-order logic,\(^{19}\) the interesting property is that the schema tuple query language \( Q \) is still decidable under complementation. In contrast, this does not hold for the \( \exists^* \forall^* \) fragment \((\neg \exists^* \forall^* \rho \equiv \forall^* \exists^* \neg \rho)\). The reason the language stays closed under complementation is intimately connected with the aforementioned schema tuple query assumption. See Minock (2003a) for details.

Discussion now splits to address two main topics. The first relates to the method and assumptions that guide the tactical query paraphraser. The second relates to the nature and adequacy of the role of the cooperative information system as a modular reasoner and the query paraphraser as a modular tactical realization component.

5.1. Query Paraphrasing

Natural language interfaces to relational databases were considered at the outset of foundational work on the relational model. In fact Codd, the founder of the relational approach, designed and partially implemented the RENDEZVOUS system which was intended to engage users in free dialogs over relational databases (Codd, 1974). Codd envisioned a system that would let users employ relatively unrestricted natural language. If the system failed to understand a users question, it would engage the user in a tight clarification dialog. Clarification dialogs where contrasted with the stroking (Weizenbaum, 1965) responses and contributive (Winograd, 1972) responses that had been implemented in the showcase dialog systems of that era. Codd proposed seven ‘steps’ or maxims that governed this ‘rendezvous’ process. Of note the fourth step involved “system re-statement of user’s query”.
Given the focus on building up the basic relational technology, it is not surprising that most work in the 1970's was of a visionary nature. Building actual NL interfaces and dialog systems over databases really started in the 1980s and into the 1990s. Of note there have been, over the years, various relational query to natural language generators (Codd, 1974; Lowden and de Roeck, 1986; Ljungberg, 1991).

Of the prior relational or SQL to natural language generators, none have adopted techniques resembling the subsumption based approach presented here. More generally none can be said to be ‘semantic’. The most advanced of the prior ‘syntactic’ generators seems to be the REMIT system (Lowden and de Roeck, 1986). Similar to the system here, REMIT maps relational calculus expressions to natural language. REMIT does this in two phases. The first phase maps the relational calculus expression to a predicate/argument structure underlying the English paraphrase. The second phase maps the predicate/argument structure to English. In the first phase a focus is determined which is the ‘root’ variable of the given query. The focus of a query in REMIT corresponds roughly to the free variable \( x \) of queries here. However since REMIT does not place restrictions on the tuple calculus expressions, the determination of focus in REMIT is not as straightforward as it is in the work here. In any case once a focus is determined REMIT follows the syntactic structure of the query and associates linguistic knowledge attached to attributes of the current focus. REMIT recursively applies the same strategy to satellite variables reached through joins.

When REMIT translates the resulting predicate/attribute structure to English, the issue of ambiguity is finessed by employing an indentation strategy that clarifies which phrases modify which conditions. The approach here, in contrast, has not addressed the issue of ambiguity. Certainly one could easily adopt the REMIT strategy, but another strategy is to mark attribute pairs over the schema which might be confused for one another. Then generation would be constrained to produce a paraphrase in which such ambiguities were avoided. Another way to finess the issue is simply to generate multiple paraphrases of the same query.

It should be said that REMIT covers projection and aggregate functions in queries, while the approach here will need to be extended to handle such constructs. On the other hand it is not clear what the configuration requirements of REMIT are, though they appear to be considerable. In contrast, the configuration of the system here is well spelled out. Finally the semantic basis of the generator here gives a very clear path toward integrating very specialized domain dependent language use into the system.

Currently it is assumed that a single, highly aggregated sentence, may describe a query. Certainly there is some limit to the number of query conditions that may be aggregated into a single sentence. Techniques to break
up the sentences must be entertained if the approach here is to scale to more complex queries. Issues such as pronominalization and ellipsis are not yet addressed in this work, but will become more important as techniques to span complex query descriptions over multiple sentences are considered.

Though the idea of using a phrasal grammar is not new (Reiter, 1990), nor is using classification in text generation (Reiter and Mellish, 1992), the approach here is new in regard to exploiting the properties of the query formation language $L$.

### 5.2. Modular architectures

This paper has described an architecture in which a cooperative information system plays the role of strategic component and a query paraphraser plays the role of a tactical component. The interchange language between these components is the language of the schema tuple queries expressed in $Q$.

While both sub-systems countenance the semantic constraints imposed by the schema and functional dependencies as well as other relational constraints, the approach is purely pipelined, with the cooperative information system making communication act commands specified in $Q$ parameters and the description system interacting with the semantical system to obtain the most succinct descriptions of ‘queries’. This is in contrast to more general techniques that explicitly plan content through complex plan operators (Moore and Paris, 1989) or schemata (McKeown, 1985) using established rhetorical theories (Mann and Thompson, 1988). The hard-coded command approach adopted here may be less flexible, but the knowledge specification task is simplified considerably.

### 6. Conclusions

Conversational access to databases, though not as active as it used to be, has historically been of significant interest (Androutsopoulos et al., 1995, 2000). There are several reasons why this area represents a good environment in which to test natural language technology in general and natural language generation technology specifically. First relational databases naturally restrict the domain of discourse to the relations within the database and, through Entity-Relationship modeling, it is often clear which tables represent nouns and which tables or foreign key references represent verbs. Additionally there has been a fair amount of work over the problem and thus there is a lot of prior work to which the present work may be compared. Though restricting attention to access to databases does not solve the full problem of evaluation, it does at least focus the work to fixed input representations and several well known techniques. Finally relational databases represent one of those technologies from computer science that have stood the test of time and have found tremendous practical application. Relational databases remain as the
principle methods for structured information. And while it is promised that XML Query and XPath are poised to open up structured information on the web, it is likely that their essentially relational cores are what will be most widely used for querying.

In summary this article has described a scalable and structured approach to generating natural language descriptions for a broad class of relational database queries. The simplicity of the phrasal approach enables clever database administrators to author the system without requiring specialized linguistic knowledge. The firm semantic basis of the approach lends a great deal of structure to the authoring process. Notably an administrator can declare their schema ‘covered’ once they have provided phrasal entries for a bounded set of simple and join conditions.

The presented approach modularly integrates with many reasoning components due to the properties of the query class for which it is designed. To illustrate this an architecture has been proposed where the query paraphrasing component serves as a tactical component while a cooperative information system serves as the strategic component.

Notes

1 Under the assumption that paintings are painted by only a single artist.

2 Thus, to obtain artists born in the year 1904, one would write \{x|\text{Artist}(x) \land x.\text{yearOfBirth} = 1904\} rather than \{(x.id,x.name,x.\text{yearOfBirth},x.\text{yearOfDeath},x.\text{country}) | \text{Artist}(x) \land x.\text{yearOfBirth} = 1904\}.

3 The purpose of showing this query is to give an idea of the limitations of the current approach. Still algorithms are being sought to decide queries with less restrictive quantifier prefixes. Logics related to the guarded fragment (Andreka et al., 1998) seem to hold out promise.

4 This limitation is true for the core logic languages described here. In practice, these languages will be extended with attribute projection and simple aggregate operations (e.g. COUNT, SUM, AVERAGE, etc.).

5 A component reference is \(z.a\), where \(z\) is a tuple variable and \(a\) is an attribute of the relation which \(z\) ranges over.

6 Either \(=,\neq,>,\geq,\leq,\lt\) or \(\leq\).

7 The term ‘non-cyclic’ means that the graph, where query variables are vertices and joins between variables are undirected edges, is acyclic.

8 Actually in the implemented system a pattern also has linguistic features such as number, definiteness, etc. However the treatment in this paper is restricted to the simpler, non feature-based approach.

9 Morphological functions do very simple things like return the plural form of a word. They are included as a way to bend text that is contained within the database.

10 Note that the meaning of these terms here does not correspond precisely to their usage in X-bar like syntax theories.

11 Simple conditions with \(>,\geq,\leq\) operators.

12 For those familiar with Entity-Relationship modeling, the goal is to generate descriptions of an entity’s participation in one-to-many and many-to-many relationships.
One-to-one relationships are skipped because of their simplicity. Note that the approach does not systematically handle either reflexive (self-joining) relationships or general conceptual relationships involving more than 2 entity types, even though $L$ admits such queries.

13 The complete version of the phrasal lexicon would require at least 186 entries; this phrasal lexicon has 47 entries.

14 See (Gaasterland et al., 1992) for a more detailed description of these techniques as well as their connection to Grice’s original maxims (Grice, 1975).

15 Only the case of condition relaxation is considered in this paper.

16 The notion of certain answers here is that of certain answers in Abiteboul and Duschka (1998) under closed-world views. Specifically an answer $\tau$ to query $q$ is certain, if it is an answer to $q$ over all complete databases where $v_1, \ldots, v_n$ are complete views.

17 A materialized view is a view paired with a set of tuples that presumably satisfy the view.

18 STEP, it should be noted, is currently built around a single sentence natural language understanding approach, though, for many applications a forms based interface is more appropriate.

19 Note that the $\exists \forall \rho$ is decidable, when $\rho$ is function free, because one does not need to introduce Skolem functions, only Skolem constants, when converting formulas to CNF (conjunctive normal form). This yields a finite Herbrand Universe, thus the resolution process will terminate after a fixed number of steps.

References


