Building A Model Query Language via SML

Yao-Chuan Tsai

Dept. of Business Administration, National Cheng Kung University
(tsaia@mail.ncku.edu.tw)

Abstract

Structured modeling language (SML) is modeling language for the structured modeling (SM) framework, which is designed to represent a wide range of models. The lack of query (retrieval) capabilities in most extant modeling systems constitutes one of the greatest anachronisms of contemporary practice. This paper develops a model query language called SMQL for SM. While previous research requires users to know the models are represented in a DBMS in order to perform queries, our approach does not. This has been achieved through the use of SML as the Model Definition Language and develop SMQL on top of an SQL-based DBMS. This approach will save tremendous effort and time for language development. Also, since SML's Natural Language Summary is a non-technical interface that enable the users easily learn the "structure" of the data, but not its technical details, the users thus can use SML's Natural Language Summary (or SML schema) for performing data queries, and can free from having to know the details of data representation.

1. Introduction

Model query is the task of retrieving information about models either before or after model manipulation [2]. Model query facilities allow users to understand their models thoroughly, to verify whether or not a model corresponds to reality, and to make inferences from a model.

Traditionally, queries about Management Science/Operations Research (MS/OR) models are for sensitivity analysis and optimization. Sensitivity analysis and optimization are applied to "what if" and "what's best" queries in management decision-making. While modeling systems provide support for the specification and computation of MS/OR models, they lack flexible retrieval capability, as in the tradition of database management systems, for querying about models themselves. The lack of such query capabilities in most extant modeling systems constitutes one of the greatest anachronisms of contemporary practice. To quote Geoffrion [10],

> *"... Data Management and flexible retrieval capability are just as important for most MS/OR applications as the functions performed by the solvers toward which the models usually are oriented."*

Structured modeling [10, 11] was developed to provide a very general approach to the problems and activities associated with modeling. SML [16, 17] is a modeling language for the structured modeling framework, which represents the semantics as well as mathematical structure of a model. This paper develops a model query language SMQL for structured modeling (SMQL has two sub-languages of EDQL and ScQL.). A model query language which, like a database query language, is a built-in facility in modeling systems that helps answer model queries on a more customized basis. A model query language does not pretend to cope with the entire universe of possible model queries, but serves as a convenient tool allowing users to easily construct their own queries. A query language has the task of expressing queries as well as being easy to use. If a model query language can be created that is both easy to use and powerful for retrieval, it would be among the most useful feature of modeling systems. However, 'ease of use' and 'great expressive power' in a language might be contradictory. For practical reasons and usability considerations, we believe that a simple and useful query language would be more acceptable than a powerful but complicated query language. This paper, hence, emphasizes simplicity of the language over expressive power. It is worthy noting that 'ease of use' is especially demanded in a today's personal computing environment.

Our approach, which proposes using an external DBMS for creating a model query language, is different from previous research that represents and queries MS/OR models within a DBMS. While previous research requires managing models within a DBMS environment, our approach does not. Instead of viewing a DBMS as a modeling system, our approach considers a model query system as a front end to a DBMS external to the system. We take two steps in order to create a model query language. First, we use SML as the model definition language. Second, we internally represent SML models in relations, and then build our model query language on top of the SQL [6] offered by a relational DBMS, by using the semantics of the SML model definitions.

By necessity, we presume that the reader is acquainted with the basic terms of SML. For example, the reader should know that an SML model consists of a schema and elemental detail tables, that a schema represents a structured model's general structure, and that elemental detail tables instantiate a model schema to a specific model instance. Figure 1 presents a model schema of the Happy Valley Food Cooerative (HVFC) example in [31]. Figure 2 presents the model's elemental detail tables.

The organization of this paper as follows. Section 2 reviews certain points about data models from the database literature for the benefit of readers who may not be familiar with these ideas. SMQL has two sub-languages of EDQL and ScQL. Section 3 develops the EDQL language for retrieving a model's elemental detailed data. Section 4 develops the ScQL language for retrieving a model's general structure. Finally, Section 5 summarizes the contributions and future research.

2. Selected Ideas from Data Models

We will borrow from the approaches of database query languages to create our model query language. Approaches to creating a database query language can be considered in two ways, one semantic and the other syntactical. The semantic aspect of a query language determines how the user imagines the data to be (e.g., network, table, etc.). The various methods (e.g., text, graphics, menu, etc.) by which the users may specify queries based on this image make up the syntactical aspect of a query language. Semantic approaches to a database query language use data models, which play an integral part of our language development. A text-based method of using key words is also adopted to create our model query language.

Semantic approaches to a query language interface use data models to represent data, and queries are phrased based on the model used. Data models are classified into 3 categories: classical data models, semantic database models, the universal relation model.

Classical Data Models

Classical data models (e.g., network, hierarchical, and relational models) have gained wide acceptance as efficient data management tools since the 1970s. The use of classical data models in data management is a significant step forward because it allows users to specify a navigational path among the abstract, 'logical' data structures when performing queries and hence removed the need for specifying access paths in a database's 'physical' storage structure. The term "logical navigation" is the process of following (or specifying) links, or more generally, relationships based on a model schema [30].

By the criterion of easy use, the relational model is no doubt superior. It provides essentially only one concept, the relation. Relational DBMSs have stressed high-level query languages, while DBMSs based on hierarchical and network models have tended to have lower-level languages [32]. Although relational database query languages are claimed as being easy to use, understanding the relational structures and supplying a logical navigational path still can be difficult tasks for many users, especially in a complex database with dozens of relations. We cannot expect every user to know the underlying relational database structure very well.

Semantic Data Models

Semantic data models (e.g., the Entity-Relationship Model [4], the Functional Data Model [27], and Hammer and McLeod's Semantic Database Model [22]) are a new generation of data models subsequent to the development of classical data models. While classical data models offer appealing data management techniques, they are rather poor at conveying the semantics of an application. For example, if information regarding an object is represented in several relations, the relational data model requires the user to think of the object by traversing from one relation to another, making it more difficult to form complex objects out of simple ones. Recognizing such limitations, semantic data models were created to provide richer, more expressive concepts with which to capture more meaning than is possible with classical data models. Because semantic data models allow users to think of data more directly than classical data models do, users can construct navigational queries more easily through data relationships on a 'semantic' database model schema than on a 'classical' data model schema [23]. Simple query languages are the natural by-products of most semantic data modeling.

The Universal Relation Model

The universal relation model also attempts semantic enhancements for the purpose of freeing users from the burdens of logical navigations among the relations of a database. The universal relation model proposed by Ullman [32] allows the user to imagine that the data of an entire database are kept in a single relation, whose schema consists of all the columns in any of the relations of the database. The user doesn't have to know the relational structure of the database, and hence need not specify a logical navigational path among the relations in a database when posing a query. To

support a universal relation user interface within a database system, a database designer has to define not only a relational database itself, but also the additional "connections" of related columns in relations. Having done so, the navigational path, not specified in a universal relation query by a user, can be specified automatically by the system. The system must therefore work harder to interpret queries on the database.

3. Elemental Detail Query Language

This section discusses the Elemental Detail Query Language (EDQL) for retrieving a model's detailed data. The development of EDQL uses an idea of semantic data models.

3.1 Approach

We have mentioned in Section 2 that semantic data models are semantically richer than the relational model. Semantic data models, therefore, enable users to think of data in ways that correlate more directly to how data arise in the world than in a relational database. Naturally, users can specify a navigational path more easily on a semantic data model schema than on a relational scheme.

An SML schema, like a semantic data model schema, is semantically richer than a relational scheme [9]. Rather than phrasing queries (using SQL or QBE) on elemental detail tables, we propose to phrase EDQL queries on 'imaginary' primitive tables (discussed in the next section) that are directly determined by an SML schema. A translation procedure would translate an EDQL query on "primitive tables" into an equivalent SQL query on "elemental detail tables". The translated SQL query can then be executed on the DBMS. Users can thus be relieved from having to know the elemental detail tables when posing queries about a model's detailed data. As a result, users only need know an SML schema of interest, not both an SML schema and the elemental detail tables.

3.2 Primitive Tables

According to Steps 1 and 2 of Geoffrion's [13] Table Structuring Procedure, an SML schema directly determines three kinds of primitive table structures. The primitive tables are genus tables, dependency tables, and symbolic parameter tables. Conceptually, genus tables contain the elemental data of genera. Dependency tables contain the detailed information of functional dependencies (FD) and multi-valued dependencies (MVD) defined in an SML schema. Symbolic parameter tables describe the details of symbolic parameters that are defined in generic rules

Primitive tables have only three kinds of columns. The first kind, called an identifier (ID) column, is the column of identifiers that corresponds to an index in SML. The second kind, called a VALUE column, holds the values of attribute, function, test elements, or symbolic parameters. The third kind, called an interpretation (INTERP) column, holds the interpretation of an identifier.

The possible forms of primitive table structures are listed below (we follow the column order convention of [13].

1. Genus tables:
   (a) ID ||
   (b) ID || INTERP
   (c) ID || VALUE, INTERP
   (d) ID, ..., ID ||
   (e) ID, ..., ID || VALUE
   (f) || VALUE
2. Dependency tables:
   (a) ID || ID
   (b) ID, ..., ID || ID
3. Symbolic parameter tables:
   (a) || VALUE
   (b) ID || VALUE
   (c) ID, ..., ID || VALUE

If column names are attached to columns of a relation, then the order of the columns becomes unimportant. However, for simplicity we intend that the primitive tables use the same column order convention as Geoffrion's Table Structuring Procedure. Hence, primitive tables are exactly the unjoined elemental detail tables (i.e., prior to Step 3 of the Procedure), and genus tables, dependency tables and symbolic parameter tables are exactly Step 1 tables, Step 2A tables and Step 2B tables, respectively in [13]

In order that the primitive table structures make sense as a conceptual tool, we propose the following mnemonic EDQL Table Naming Conventions for primitive tables.

Primitive Table Naming Conventions:

1. **Genus Table Naming Convention** Each genus table name coincides with the generic name associated with the genus whence it derives. Each ID column name coincides with the corresponding index. The VALUE column name is simply "value". The INTERP column name is simply "interp".
2. **Dependency Table Naming Convention** Each dependency table name coincides with the functional or multi-valued dependency (e.g., h1(j), k2*(i, j), etc.). Each independent ID column name coincides with the corresponding independent index. The dependent ID column name coincides with the corresponding functional or multi-valued dependency name (e.g., h1, k2*, etc.).
3. **Symbolic Parameter Table Naming Convention** Each symbolic parameter table name is the stem of the symbolic parameter, to which is appended an underscore and the generic name of the associated genus whence it derives. Each ID column name coincides with the corresponding index. The VALUE column name is simply "value".

The HVFC model's primitive tables using EDQL Table Naming Conventions are listed in Figure 3.

Primitive table structures are directly determined by an SML schema. One may speak without ambiguity of "the" primitive table corresponding to any given genus, of "the" primitive table corresponding to any given functional dependency or multi-valued dependency, and of "the" primitive table corresponding to any given symbolic parameter. Figure 4 shows the correspondences of a model's primitive tables to genera, functional/multi-valued dependencies or symbolic parameters that are defined in an SML schema. For example, the primitive table **MEMm** corresponds to the genus **MEM** defined in the HVFC model schema's second paragraph, the primitive table **m1(o)** corresponds to the functional dependency **m1** defined in the **ORD** paragraph, etc.

It should be noted that we are not proposing that the data should be stored in primitive tables; just that users should be allowed to perceive the data as if they were stored that way. By Step 3 of the Table Structuring Procedure [13], elemental detail tables are joined from primitive tables (Step 1 tables, Step 2A tables and Step 2B tables) without changing any data. Hence, primitive tables are equivalent to elemental detail tables. If Step 3 is not performed, then primitive tables are exactly elemental detail tables.

Because the structures of primitive tables are directly determined by an SML schema, primitive tables are easy to imagine, and thus facilitate phrasing a query about a model's detailed data, whether Step 3 is performed or not. Informally, data are stored in elemental detail tables. Primitive tables are imaginary and do not actually exist. EDQL queries use primitive table structures, not elemental detail table structures. An EDQL query has this simple form:

SELECT table.column, …
WHERE conditions

where 'table.column' uses Primitive Table Naming Conventions, and 'conditions' are the imposed constraints on the information retrieved. EDQL queries are similar to SQL queries, but do not have a FROM clause. The information regarding FROM tables is embedded in the associated SELECT clauses. An EDQL query will be translated into an equivalent SQL query.

3.3 EDQL Query Translation

This section discusses a procedure for translating an EDQL query on primitive tables into an equivalent SQL query on elemental detail tables. The translated SQL query can then be executed on the DBMS.

Note that each primitive table corresponds to exactly one elemental detail table, but not vice versa. Figure 5 lists the correspondence between the HVFC model primitive tables and elemental detail tables. For example, the primitive table

MEMm corresponds to the elemental detail table **MEM**,but the elemental detail table **MEM** corresponds to the primitive tables **MEMm**, **MNAMEm**, **MADDRm**, and **BALm**.

Also, each column in the primitive tables corresponds to exactly one elemental detail table's column, but not vice versa. Figure 6 lists the column correspondence between the HVFC model primitive tables and elemental detail tables. For example, the **m** column of the primitive table **MEMm** corresponds to **MEM** column of the elemental detail table **MEM**, but the **MEM** column of the elemental detail table **MEM** corresponds to four primitive tables' columns, which are the **m** column of the primitive table **MEMm**, the **m** column of the primitive table **MNAMEm**, the **m** column of the primitive table **MADDRm**, and the **m** column of the primitive table **BALm**.

Because each primitive table corresponds to exactly one elemental detail table and each column in the primitive

tables corresponds to exactly one elemental detail table's column, an EDQL query on primitive tables can be translated into an SQL query on elemental detail tables without any ambiguity. We will show that the EDQL translation procedure would unambiguously translate an EDQL query on primitive tables into an SQL query on elemental detail tables. The translated SQL query could then be executed on the DBMS.

EDQL Translation Procedure:

Input: An EDQL query on primitive tables
Output: An SQL query on elemental detail tables
Procedure:

Step 1: (Change to the SELECT-FROM-WHERE pattern) Translate the query's 'SELECT-WHERE' pattern into the 'SELECT-FROM-WHERE' pattern by creating a FROM clause right after each SELECT clause, but before the SELECT command's WHERE clause. The SELECT and WHERE clauses do not change. Each newly created FROM clause has the form of "FROM t1, t2, ...". t1, t2, ... are primitive table names which are exactly the unique collection of the `table' names in the associated SELECT clause's 'table.column' names of the EDQL query.

Step 2: (Change all 'table.column' names in SELECT and WHERE clauses) Replace each primitive table's 'table.column' name with a corresponding elemental detail table's 'table.column' name in all SELECT and WHERE clauses.

Step 3: (Change all 'table' names in FROM clauses) Replace each primitive table name with a corresponding elemental detail table name in all FROM clauses.

Step 4: (Delete duplicate elemental detail table names in each FROM clause) For each FROM clause, if there is a duplicate elemental detail table name in the FROM clause after Step 3, delete it from the clause. Stop.

The following proposition shows that the EDQL translation procedure unambiguously translates an EDQL query on primitive tables into an SQL query on elemental detail tables.

Proposition 1 *The EDQL translation procedure uniquely translates to an SQL query on elemental detail tables from an EDQL query on primitive tables.*

Proof:

We shall prove that there is no ambiguity in translating an EDQL query on primitive tables into an SQL query on elemental detail tables.

Remember that EDQL queries have no FROM clause, but SQL queries must have a FROM clause. Step 1 transforms the EDQL query's SELECT-WHERE pattern into the SQL's SELECT-FROM-WHERE pattern. The primitive table names of each FROM clause are unique and are exactly the collection of the 'table' names in the associated SELECT clause's 'table.column's of the EDQL query.

Since each column in the primitive tables identifies exactly one elemental detail table column, Step 2 would uniquely translate each primitive table's 'table.column' name in SELECT and WHERE clauses to a corresponding elemental detail table's 'table.column' name. Furthermore, since each primitive table identifies exactly one elemental detail table, Step 3 would uniquely translate each primitive table name in FROM clauses to a corresponding elemental detail table name. Hence, the translated SQL query on elemental detail tables must be unique. ■

An example of the HVFC model queries in EDQL and SQL is given below.

Example 1 *To print suppliers, addresses, and offering prices of all suppliers that supply Granola (GR) in the HVFC model*, we write

```
SELECT SADDRs.s, SADDRs.value, PRICEsi.value
WHERE SADDRs.s = PRICEsi.s
        AND
        PRICEsi.i = 'GR'
```

The EDQL Translation Procedure is applied below. Step 1 of the EDQL Translation Procedure creates the clause 'FROM PRICEsi' after the SELECT clause. Step 2 replaces the primitive table's column names PRICEsi.s, PRICEsi.i, and PRICEsi.value with the elemental detail table's column names OFFER.SUP, OFFER.ITEM, and OFFER.PRICE respectively. Step 3 replaces the primitive table name PRICEsi with the corresponding elemental detail table name OFFER. Step 4 is not applied because there is no duplicate table name in the query's FROM clause after Step 3. As a result, the EDQL query is translated into this SQL query by the EDQL translation procedure:

```
SELECT SUP.SUP, SUP.SADDR, OFFER.PRICE
FROM SUP, OFFER
```

WHERE SUP.SUP = OFFER.SUP
AND
OFFER.ITEM = 'GR' ■

4. ScQL Query Language

This section discusses a universal relation query language, which we call Schema Query Language (ScQL), for retrieving a model's general structure. ScQL has similar functions to a data dictionary in databases. The ability of users to ask questions about the general structure of a database is an important aspect of intelligent query systems [19].

4.1 Approach

Since we will use a relational DBMS for creating a MQL, we shall represent an SML schema in relations ("schema tables") in order to query general model structure. The query language of the adopted relational DBMS would then be applicable. But it is possible to advance to an even simpler language interface by simulating a universal relation [32]. This technique requires proper design of schema tables, a translation procedure, and some assumptions. Schema tables contain the schema data for retrieval. The translation procedure would use our assumptions and would translate a user's ScQL query (universal relation query) into an SQL query on schema tables. The translated SQL query could then be executed on the DBMS.

4.2 Schema Tables

This section discusses the representation of an SML schema in relations (tables). Schema tables represent an SML schema. The Internal Representation Tables (IRTs) in FW/SM [26] represent one kind of schema tables. IRTs, however, are intended for storing the schema information for purposes other than for query processing. Instead of using IRTs, this section proposes a different kind of schema tables that can achieve a simple query interface.

The first step in order to create schema tables is to define the column names of the tables. We propose the column names shown in Figure 7. The meaning of those column names are obvious to anyone who knows SML.

Next, in order to create useful table structures, we investigate the relationships among the defined columns. From the definitional semantics of an SML schema, the predicates listed below describe the relationships among the "columns" specified in Figure 7. Each predicate defines a functional or multi-valued dependency relationship of the involved columns. Let "X → Y" denote "an X value determines a Y value" and "X —>> Y" denote "an X value determines a set of Y values".

1. Each genus has a genus type. Also, consider a modular paragraph as a paragraph of type /m/.
   The predicate is "*TYPE is the type of NAME.*"
   NAME → TYPE.
2. Each non /pe/ genus has a calling sequence.
   The predicate is "*CALLING_SEQUENCE is the calling sequence of NAME.*"
   NAME → CALLING_SEQUENCE.
3. Each genus has an optional domain statement.
   The predicate is "*DOMAIN is the domain statement of NAME*."
   NAME → DOMAIN.
4. Each /f/ or /t/ genus has a generic rule.
   The predicate is "*RULE is the generic rule of NAME*."
   NAME → RULE.
5. Each /a/ or /va/ genus has an optional range statement.
   The predicate is "*RANGE is the range statement of NAME.*"
   NAME → RANGE.
6. Each genus has an optional index set statement.
   The predicate is "*ISS is the index set statement of NAME.*"
   NAME → ISS.
7. Each paragraph has an optional interpretation.
   The predicate is "*INTERPRETATION is the interpretation of NAME.*"
   NAME → INTERPRETATION.
8. Each paragraph has an indentation level.
   The predicate is "INDENTATION is the indentation of NAME."
   NAME → INDENTATION.
9. Each paragraph has a sequence number.
   The predicate is "*PARAGRAPH# is the sequence number of NAME.*"

NAME → PARAGRAPH#.

10. Each genus is either indexed or unindexed.
    The predicate is "*INDEXED? states whether NAME is indexed.*"
    NAME → INDEXED?.
11. Each indexed genus is either self-indexed or externally indexed.
    The predicate is "*SELF-INDEXED? states whether NAME is self-indexed.*"
    NAME → SELF-INDEXED?.
12. Each self-indexed genus introduces a new index.
    The predicate is "*NEW_INDEX is the index introduced by NAME.*"
    NAME → NEW_INDEX.
13. Each self-indexed genus may have an alias index.
    The predicate is "*ALIAS_INDEX is the alias index for NAME.*"
    NAME → ALIAS_INDEX.
14. Each indexed genus may have a symbolic index tuple.
    The predicate is "*INDEX_TUPLE is the index tuple for NAME.*"
    NAME → INDEX_TUPLE.
15. A non /pe/ genus calls some genera in the calling sequence.
    The predicate is "*NAME is a called genus of CALLING_NAME.*"
    CALLING_NAME —>> NAME.
16. A non /pe/ genus has a sequence of components in a calling sequence.
    Two predicates are "COMPONENT is a called component of CALLING_NAME" and "COMPONENT# is a component's sequence number of CALLING_NAME".
    CALLING_NAME —>> COMPONENT.
    CALLING_NAME —>> COMPONENT#.
17. Each dependency is either a functional dependency or a multi-valued dependency.
    The predicate is "*FD? states whether DEPENDENCY_NAME is a functional dependency.*"
    DEPENDENCY_NAME → FD?.
18. Each dependency has a dependency name.
    The predicate is "*The name of DEPENDENCY is DEPENDENCY_NAME.*"
    DEPENDENCY_NAME → DEPENDENCY.
19. There are functional or multi-valued dependencies used for the first time in the generic calling sequence of some genus paragraphs.
    The predicate is "*DEPENDENCY_NAME is the dependency used for the first time in the genus paragraph NAME.*"
    DEPENDENCY_NAME → NAME.
20. There are distinct symbolic parameters used in the generic rule of a genus paragraph.
    The predicate is "*PARAMETER is the symbolic parameter used in the generic rule of genus paragraph NAME.*"
    PARAMETER → NAME.
21. A module has a sequence of sons.
    Two predicates are "*SON is the son of NAME" and "SON# is the son's sequence number within NAME.*"
    NAME —>> SON,
    NAME —>> SON#.
22. A module or genus can have defined key phrases.
    The predicate is "*KP is a defined key phrase of NAME.*"
    NAME —>> KP.
23. A module or genus paragraph can have referenced key phrases in its interpretation.
    The predicate is "*REFERENCED_KP is a referenced key phrase of NAME.*"
    NAME —>> REFERENCED_KP.

Based on the dependency relationships established in the above predicates, we propose the following **Schema Table Structures** for our ScQL development. We aggregate columns into the same table as much as possible in order to avoid data redundancy. Aggregating columns also avoids unnecessary join operations when answering a question, and hence can facilitate computing a query.

We follow the convention of drawing a double vertical line just to the right of each key in the schema table structures. A key of a relation is a column or a set of columns whose values uniquely identify each tuple in the relation.

1. Table Name: *PARAGRAPH_TBL*
   NAME || INDEXED?, INDEX_TUPLE, SELF-INDEXED?, NEW¥_INDEX, ALIAS_INDEX, CALLING_SEQUENCE, TYPE, ISS, DOMAIN, RANGE, RULE, INTERPRETATION, INDENTATION, PARAGRAPH#

When the data are loaded into the **PARAGRAPH_TBL** table, they should obey the following rules: (1) Each row of the table corresponds to an SML schema's paragraph. (2) For a row of a NAME (i.e., genus or module), (2a) if a non-NAME column is optional in SML (e.g., a range statement is optional for a /a/ genus), and is not specified in the SML schema, then use "NULL" as data entry in the table, and (2b) if a non-NAME column is not allowed by SML (e.g., a calling sequence is not allowed for a /pe/ genus), then use "NA" as the data entry in the table.

2. Table Name: *CALL_TBL*
   CALLING_NAME, NAME || COMPONENT, COMPONENT#
3. Table Name: *DEP_TBL*
   DEPENDENCY_NAME || FD?, DEPENDENCY, NAME
4. Table Name: *PARAMETER_TBL*
   PARAMETER || NAME
5. Table Name: *TREE_TBL*
   NAME, SON || SON#
6. Table Name: *KP_TBL*
   NAME, KP ||
7. Table Name: *KP_REF_TBL*
   NAME, REFERENCED_KP||

The above schema tables have two properties. First, all the column names, except NAME, are unique across all schema tables. Second, the column name NAME is common among the schema tables and serves to join the different tables for multiple table queries. We will use these two properties in translating queries in the next section.

ScQL supports the universal relation interface. This means that users need not know the structure of the schema tables. Instead, for query purposes, users can imagine schema tables as a single table that contains all the columns listed in Figure 7. As a result, users need not specify a navigational path (no nested queries) when posing a query about a model schema.

An ScQL query has the following simple form:

SELECT column, …
WHERE conditions

4.3 Query Computation and Confirmation

This section gives a procedure for translating an ScQL query on the universal relation into an equivalent SQL query on schema tables. The translated SQL query could then be executed on the DBMS. We also discuss a confirmation approach for resolving possible ambiguities.

Remember that our schema tables have two properties. First, all the column names, except NAME, are unique across all schema tables. Second, the column name NAME is common among the schema tables and serves to join the different tables for multiple table queries.

Further, when translating an ScQL query into an SQL query, the ScQL processor assumes that the user intends to retrieve all genus/module names if an ScQL query has the form "SELECT NAME" without any imposed condition.

Because of its two properties and the assumption mentioned above, schema tables which are required for answering a given ScQL query could be uniquely determined by the specified column names, except for column name NAME. A universal relation view is then feasible for building ScQL. The user thus could imagine the schema tables as a universal relation and could pose a simple ScQL query without the burden of having to know the structure of the schema tables and specify a navigational path.

The following procedure will translate an ScQL query into an SQL query on schema tables.

ScQL Translation Procedure:

Input: An ScQL query on a universal relation which has the collection of all columns in Figure 7.
Output: An SQL query on schema tables

Procedure:

Step 1: If the query is "SELECT NAME", then the translated SQL query is

```
SELECT NAME
FROM PARAGRAPH_TBL
```

Otherwise, go to Step 2.

Step 2: If the query has a "WHERE conditions" clause, then translate the query into an SQL query of the following form

SELECT C1, C2, …
FROM T1, T2, …
WHERE conditions

Otherwise, the translated SQL query has the following form,

SELECT C1, C2, …
FROM T1, T2, …

The translations are subject to the following rules.

2.a. The **SELECT** and **WHERE** clauses of the translated SQL query are the same as those in the ScQL query.

2.b. The **FROM** clause is determined as follows:

For each column name **Ci** specified in the **SELECT** and **WHERE** clauses of an ScQL query,

2.b.1 If **Ci** is **NAME** then skip Ci.

2.b.2 If **Ci** is not **NAME**, then find table **Tj** that contains the column name **Ci.** (Remember, all the column names except NAME are unique in the schema tables.)

Stop.

We shall prove that there is no ambiguity in translating an ScQL query into an equivalent SQL query on schema tables.

Proposition 2 *The ScQL translation procedure uniquely translates an ScQL query into an SQL query on schema tables.*

Proof:

We shall prove that there is no ambiguity in translating an ScQL query into an SQL query on schema tables. An ScQL query has one of the following two forms.

SELECT C1, C2, …

or

SELECT C1, C2, …
WHERE conditions

If the query is "**SELECT NAME**", Step 1 would create the SQL query:

SELECT NAME
FROM PARAGRAPH_TBL

Otherwise, Step 2 would create an SQL query of the form

SELECT C1, C2, …
FROM T1, T2, …
WHERE conditions

or

SELECT C1, C2, …
FROM T1, T2, …

The SELECT and WHERE clauses are not changed by Step 2.a. The FROM clause is determined by Steps 2.b.1 and 2.b.2 as explained below.

Step 2.b.1 ignores the column name, NAME. Step 2.b.1 is safe because the query is not "**SELECT NAME**" and the FROM table names can be determined by other column names in the query, using Step 2.b.2.

Since all column names except NAME are unique among schema tables, Step 2.b.2 must uniquely identify the table name **Tj** that contains the column name **Ci** which is not 'NAME'. ■

We have shown that the ScQL Translation Procedure uniquely computes an SQL query on schema tables. However, because ScQL uses a universal relation interface, and because the translation from an ScQL query into a query on schema tables appears to be "black magic" to users, some users might be concerned that the navigation computed by the ScQL translator might not match the one intended by the user [20]. We can use the confirmation approach to resolve the possible ambiguities [30]. That is, in addition to giving a system-computed answer, the system should give an explicit query interpretation in natural language for the user's confirmation. If the ScQL query computation does not respond exactly as was intended, the user should use SQL to specify his intended, complicated query.

Note that each column name, except NAME, uniquely identifies a predicate in Section 4.2. If the query is **SELECT NAME**, the query interpretation would be the default statement to "List all genus and module names" (i.e., the query assumed by the ScQL Translation Procedure); otherwise, the query interpretation would be a collection of the predicates identified in the query by the specified column names, except NAME.

We present an illustrative example which shows the translation from an ScQL query to an SQL query, and the

confirmation of the query.

Example 2 *To print all indexed entity genera and their key phrases in the Happy Valley Food Coop (HVFC) model schema*, we write the ScQL query below.

```
SELECT NAME, KP
WHERE (INDEXED? = 'Y')
        AND
(TYPE = 'pe' OR 'ce')
```

We apply the ScQL Translation Procedure to translate the ScQL query into an SQL query on schema tables. The translated SQL query can then be executed on a DBMS. Since the above query is not "**SELECT NAME**", Step 2, not Step 1, of the ScQL Translation Procedure is applied. Columns INDEXED?, TYPE, and KP correspondingly determine tables PARAGRAPH_TBL and KP_TBL. The translated SQL query is listed below.

```
SELECT NAME, KP
FROM PARAGRAPH_TBL, KP_TBL
WHERE (INDEXED? = 'Y')
        AND
      (TYPE = 'pe' OR 'ce')
```

Besides the procedure's translation and the execution on a DBMS, the ScQL system would identify the following predicates in the query interpretation, based on the column names KP, INDEXED?, and TYPE in the ScQL query.

"KP is a key phrase of NAME."
"INDEXED? states whether NAME is indexed or not."
"TYPE is the type of NAME."

The user can confirm whether or not the ScQL query and the query interpretation match his intention to "*print all indexed entity genera and their key phrases"*. The combination of the ScQL query and the query interpretation implies "*to print NAME and the associated KP, where NAME is indexed and has a type of /pe/ or /ce/*, which matches the intended query. ■

5. Conclusion

We argue that there is a need for model queries in MS/OR as well as other fields. This paper develops a model query languages SMQL. SMQL has two sublanguages of EDQL and ScQL. EDQL allows users to specify a query in terms of a semantic SML schema. Because an SML schema is semantically richer than a relational scheme, users can specify a navigation path more easily in terms of an SML schema than in terms of the relational structures of elemental detail tables. The purpose of ScQL, like that of SEDQL, is to provide users with a very simple interface so that they can easily retrieve information about a model's general structure without having to specify a navigational path. When users cannot confirm the complicated queries using ScQL, they may either use SQL or simply browse the SML schema.

SMQL is an example of a useful model query language. Other model languages may well lend themselves to the development of a useful model query language if they are semantic in nature and support representational independence of general model structure and detailed data. Although text-based SMQL may be successful as a model query language, the potential of graphics, menu, voice, mice and other user interface styles should also be examined.

SMQL is a retrieval language for structured modeling. Future research should also be done on a model manipulation language that would allow users to edit a structured model's general structure and the detailed data in a convenient way. One approach, in the context of structured modeling, would be to build such a language based on schema operations [29] and a smart loader/editor [13].

References

[1] ANSI, American National Standards Institute: Database Language SQL, Document ANSI X3.135 (1986). Also available as International Standards Organization Document ISO/TC97/SC21/WG3 N117.

[2] W.F. Burger, MLD: A Language and Data Base for Modeling, IBM Research Division, San Jose, Research Report RC 9639 (#42311), (September 1982).

[3] S. Chari, *Knowledge Representation Using Structured Modeling*, Ph.D. Thesis, Anderson Graduate School of Management, UCLA, (1988).

[4] P.P. Chen, The Entity-Relationship Model - Toward a Unified View of Data, ACM Transaction on Database Systems, 1:1 (March 1976), 9-36.

[5] C. J. Date, *An Introduction to Database Systems*, Volume 1, Fourth Edition, Addison-Wesley, MA, (1986).

[6] C. J. Date, *A Guide to the SQL Standard*, Addison-Wesley, MA, (June 1987).

[7] D. R. Dolk, Data as Models: An Approach to Implementing Model Management, *Decision Support Systems*, 2:1 (March, 1986), 73-80.

[8] D. R. Dolk, Model Management and Structured Modeling: The Role of an Information Resource Dictionary System, *Communications of the ACM*, 31:6 (June, 1988), 704-718.

[9] C. K. Farn, *An Integrated Information System Architecture Based on Structured Modeling*, Ph.D. Thesis, Anderson Graduate School of Management, UCLA, (1985).

[10] A. M. Geoffrion, An Introduction to Structured Modeling, *Management Science*, 33:5 (May, 1987), 547-588.

[11] A. M. Geoffrion, The Formal aspects of Structured Modeling, *Operations Research*, 37:1 (January-February, 1989), 30-51.

[12] A. M. Geoffrion, Model Queries, Informal Note, Western Management Science Institute, Anderson Graduate School of Management, UCLA, (March, 1989).

[13] A. M. Geoffrion, *SML: A Model Definition Language for Structured Modeling,* Working Paper 360, Western Management Science Institute, Anderson Graduate School of Management, UCLA, (August, 1990).

[14] A. M. Geoffrion, *A Library of Structured Models*, Informal Note, 275 pages, Anderson Graduate School of Management, UCLA, (1990).

[15] A. M. Geoffrion, FW/SM: A Prototype Structured Modeling Environment, *Management Science*, 37:12 (December, 1991), 1513-1538.

[16] A. M. Geoffrion, The SML Language for Structured Modeling: Levels 1 and 2, *Operations Research*, 40:1 (January-February, 1992), 38-57.

[17] A. M. Geoffrion, The SML Language for Structured Modeling: Levels 3 and 4, *Operations Research*, 40:1, (January-February 1992), 58-75.

[18] A. M. Geoffrion, Structured Modeling: Survey and Future Research Directions, ORSA CSTS Newsletter, 15:1 (Spring, 1994).

[19] M. Jarke and Y. Vassiliou, A Framework for Choosing a Database Query Language, ACM Comput. Surveys, 17:3 (September, 1985), 313-340.

[20] W. Kent, Consequences of Assuming a Universal Relation, *ACM Transactions on Database Systems*, 6:4 (December, 1981), 539-556.

[21] Greenblatt, D and J. Waxman, A Study of Three Database Query Languages in *Database: Improving Usability and Responsiveness*, (B. Schneiderman, ed.) American Press, New York, (1978).

[22] Hammer, M. and D. McLeod, Database Description with SDM: A Semantic Database Model, *ACM Trans. Database Syst.*, 6:3 (September 1981), 351-386.

[23] Hull, R. and R. King , Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys*, 19:3 (September 1987).

[24] R. Krishnan and D. A. Kendrick, A Knowledge-Based System for Production and Distribution Economics, *Computer Science in Economics and Management*, (1988), 53-72.

[25] M. L. Lenard, Representing Models as Data, *Journal of Management Information Systems*, 2:4 (1986), 36-48.

[26] L. Neustadter, A. Geoffrion, S. Maturana, Y. Tsai and F. Vicuña, The Design and Implementation of a Prototype Structured Modeling Environment, *Annals of Operations Research*, 38 (1992), 453-484.

[27] Shipman, D., The Functional Data Model and the Data Language DAPLEX, *ACM Trans. Database Syst*., 6:1 (March 1981), 140-173

[28] J. F. Sowa, Conceptual Graphs for a Database Interface, *IBM J. Res. Develop.*, 20, (July, 1976), 336-357.

[29] Y. Tsai, Model Integration Using SML, Decision Support Systems, 22 (1998), 355-377.

[30] Y. Tsai, The UR Data Model Revisited, *International Journal of Information and Management Sciences*, 10:1 (March, 1999).

[31] J. D. Ullman, Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems, (Computer Science Press, Rockville, MD., 1988).

[32] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume II, (Computer Science Press, Rockville, MD., 1989).

[33] W. M. Waite and G. Goos, Compiler Construction, Springer-Verlag, New York, (1984).

&MEMBERS The MEMBER SECTION of the database.

MEM m /pe/ There is a list of current MEMBERS.

MNAME (MEM m) /a/ {MEM}: Char Every MEMBER has a MEMBER NAME.

MADDR (MEM m) /a/ {MEM}: Char Every MEMBER has a MEMBER ADDRESS.

BAL (MEM m) /a/ {MEM} Every MEMBER has a BALANCE.

ITEM i /pe/ There is a list of ITEMS offered for sale.

&SUPPLIERS The SUPPLIER SECTION of the database.

SUP s /pe/ There is a list of current SUPPLIERS.

SNAME (SUP s) /a/ {SUP}: Char Every SUPPLIER has a SUPPLIER NAME.

SADDR (SUP s) /a/ {SUP}: Char Every SUPPLIER has a SUPPLIER ADDRESS.

OFFER (SUP s, ITEM i) /ce/ Select {SUP} X {ITEM} Each SUPPLIER offers to sell certain ITEMS; the list of possibilities is known collectively as the OFFERINGS.

PRICE (OFFER si) /a/ {OFFER}: Real+ Each OFFERING has its PRICE in dollars per unit quantity.

&ORDERS The ORDER SECTION of the database.

ORD o (MEM m1(o), ITEMi1(o)) /ce/ There is a list of ORDERS, each from one MEMBER for one ITEM. (Use "order number" as the identifier.)

QTY (ORD o) /a/ {ORD}: Real+ Every ORDER has an ORDER QUANTITY.

Figure 1: HVFC Model Schema (from [17])

MEM

MEM	MNAME	MADDR	BAL
BB	Brooks, B.	7 Apple Rd.	10.50
WF	Field, W.	43 Cherry La.	.00
RR	Robin, R.	12 Heather St.	-123.45
WH	Hart, W.	65 Lark Rd.	-43.00

ITEM

ITEM	INTERP
CU	Curds
GR	Granola
LE	Lettuce
SS	Sunflower Seeds
WH	Whey
UF	Unbleached Flour

SUP

SUP	SNAME	SADDR
SUN	Sunshine Produce	16 River Street
PUR	Purity Foodstuffs	180 Industrial Rd.
TAS	Tasti Supply Co.	17 River Street

OFFER

SUP	ITEM	PRICE
PUR	CU	.80
PUR	GR	1.25
PUR	UF	.65
PUR	WH	.70
SUN	GR	1.29
SUN	LE	.89
SUN	SS	1.09
TAS	LE	.79
TAS	SS	1.19
TAS	WH	.79

ORD

ORD	MEM~m1	ITEM~i1	QTY
1	BB	GR	5
2	BB	UF	10
3	RR	GR	3
4	WF	WH	5
5	RR	SS	2
6	RR	LE	8

Figure 2: HVFC Model Elemental Detail Tables (from [17])

MEMm

m
BB
WF
RR
WH

MNAMEm

m	value
BB	Brooks, B.
WF	Field, W.
RR	Robin, R.
WH	Hart, W.

MADDRm

m	value
BB	7 Apple Rd.
WF	43 Cherry La.
RR	12 Heather St.
WH	65 Lark Rd.

BALm

m	value
BB	10.50
WF	.00
RR	-123.45
WH	-43.00

ITEMi

i	interp
CU	Gurds
GR	Granola
LE	Lettuce
SS	Sunflower Seeds
WH	Whey
UF	Unbleached Flour

SUPs

s
SUN
PUR
TAS

SNAMEs

s	value
SUN	Sunshine produce
PUR	Purity Foodstuffs
TAS	Tasti Supply Co.

SADDRs

s	value
SUN	16 River Street
PUR	180 Industrial Rd.
TAS	17 River Street

OFFERsi

s	i
PUR	CU
PUR	GR
PUR	UF
PUR	WH
SUN	GR
SUN	LE
SUN	SS
TAS	LE
TAS	SS
TAS	WH

PRICEsi

s	i	value
PUR	CU	.80
PUR	GR	1.25
PUR	UF	.65
PUR	WH	.70
SUN	GR	1.29
SUN	LE	.89
SUN	SS	1.09
TAS	LE	.79
TAS	SS	1.19
TAS	WH	.79

ml(o)

0	m1
1	BB
2	BB
3	RR
4	WH
5	RR
6	RR

il(o)

0	i1
1	GR
2	UF
3	GR
4	WH
5	SS
6	LE

ORDo

o
1
2
3
4
5
6

QTYo

o	value
1	5
2	10
3	3
4	5
5	2
6	8

Figure 3: The HVFC Model's Primitive Tables Using EDQL Table Naming Conventions

Primitive Tables	Genera, FDs/MVDs, Symbolic Parameters
MEMm	Genus MEM
MNAMEm	Genus MNAME
MADDRm	Genus MADDR
BALm	Genus BAL
ITEMi	Genus ITEM
SUPs	Genus SUP
SNAMEs	Genus SNAME
SADDRs	Genus SADDR
OFFERsi	Genus OFFER
PRICEsi	Genus PRICE
ml(o)	Functional Dependency m1
il(o)	Functional Dependency il
ORDo	Genus ORD
QTYo	Genus QTY

Figure 4: Correspondence Between HVFC Model Primitive Tables and Schema Defined Genera, FD/MVDs, and Symbolic Parameters

Primitive Tables	Elemental Detail Tables
MEMm	MEM
MNAMEm	MEM
MADDRm	MEM
BALm	MEM
ITEMi	ITEM
SUPs	SUP
SNAMEs	SUP
SADDRs	SUP
OFFERsi	OFFER
PRICEsi	OFFER
ml(o)	ORD
il(o)	ORD
ORDo	ORD
QTYo	ORD

Figure 5: Correspondence Between HVFC Model Primitive Tables and Elemental Detail Tables

Primitive Table.Column	Elemental Detail Table.Column
MEMm.m	MEM.MEM
MNAMEm.m MNAMEm.value	MEM.MEM MEM.MNAME
MADDRm.m MADDRm.value	MEM.MEM MEM.MADDR
BALm.m BALm.value	MEM.MEM MEM.BAL
ITEMi.i ITEMi.interp	ITEM.MEM ITEM.INTERP
SUPs.s	SUP.SUP
SNAMEs.s SNAMEs.value	SUP.SUP SUP.SNAME
SADDRs.s SADDRs.value	SUP.SUP SUP.SADDR
OFFERsi.s OFFERsi.i	OFFER.SUP OFFER.ITEM
PRICEsi.s PRICEsi.i PRICEsi.value	OFFER.SUP OFFER.ITEM OFFER.PRICE
ml(o).o ml(o).ml	ORD.ORD ORD.MEM~ml
il(o).o il(o).il	ORD.ORD ORD.ITEM~il
ORDo.o	ORD.ORD
QTYo.o QTYo.value	ORD.ORD ORD.QTY

Figure 6: Column Correspondence Between HVFC Model Primitive Tables and Elemental Detail Tables ("X.Y" means the "Y" column in the "X" table)

Column	Meaning
NAME	Genus/module name (paragraph name)
TYPE	m, pe, ce, a, va, f, t
CALLING_SEQUENCE	Calling sequence
DOMAIN	Domain statement
RULE	Generic rule
RANGE	Range statement
ISS	Index set statement
INTERPRETATION	Interpretation
INDENTATION	Paragraph's indentation level
PARAGRAPH#	Sequence number of paragraphs
INDEXED?	Is the genus indexed? (Y/N)
SELF_INDEX?	Is the genus self-indexed? (Y/N)
NEW_INDEX	Introduces new index
ALIAS_INDEX	Alias index
INDEX_TUPLE	Generic index tuple
CALLING_NAME	Calling genus name
COMPONENT	Components in a calling sequence
COMPONENT#	Component's sequence position in a calling sequence
DEPENDENCY_NAME	Functional/Multi-valued dependency name
FD?	Is the dependency a functional dependency? (Y/N)
DEPENDENCY	A dependency (e.g., il(j), k2*(i, j))
SON	The immediate descendents of a module
SON#	Denotes the son's sequence position of a module
KP	Defined key phrase
REFERENCED_KP	Possible referenced key phrase
PARAMETER	Symbolic parameter

Figure 7: Columns of Schema Tables