INTEGRATION OF DATABASE SYSTEMS AND SMALLTALK

B. Czejdo  
Math Sciences Department  
Loyola University  
New Orleans, LA 70118

M.C. Taylor  
Computer Science Department  
University of Houston  
Houston, TX 77204-3475

ABSTRACT

We present an object-oriented data model, that allows uniform specification of database requests and application programs. The user interface is based on Smalltalk, and the object-oriented data model is represented in terms of classes and messages. Techniques are discussed for implementing such a model on top of an underlying relational database system. Those parts of application programs that cannot be translated into a relational language are handled by a Smalltalk processor. The semantics of database requests is defined in terms of a meta-model and meta-messages, using an object-oriented approach. Hence we derive rules for translation of database requests into SQL queries over a binary relational view.

1. INTRODUCTION

Merging general-purpose programming language systems with those of database systems is one of the most important areas for research [1-10]. The major problem is to alleviate the impedance mismatch [10] between these systems, arising when information must pass between two (syntactically and semantically) different languages. The GemStone system [8] is one of the leading implementations, and provides an object-oriented database language called OPAL which is used for data definition, data manipulation and general computation. It is based on the Smalltalk language [5], that is a popular object-oriented programming language and whose implementation is well understood by using a conceptual abstraction called the virtual machine. Using Smalltalk they avoid the problem of impedance mismatch because the principal concepts of Smalltalk, such as object, message and class, were adopted for their system. To meet the requirements of a database system, several enhancements were included, such as support for a multi-user disk-based environment and a large object space.

In this paper we shall describe an object-oriented database language for data definition, data manipulation and general computation, that follows the basic directions of the GemStone project. The problem of impedance mismatch is handled by using a uniform language for designing application programs. The major difference from previous work is that an underlying relational database can be used, and appropriate parts of application programs are then translated into expressions in an extended SQL language. With this approach, support for multi-user environment, large object space and optimization is provided by an underlying database management system. Some operators of the proposed language are similar to those defined in [9], but these were significantly restructured to fit the Smalltalk environment and several new operators were created to make query specification more natural for the Smalltalk programmer.

The main contributions of this paper are:

- a uniform language for expressing database requests and application programs
- using a meta-model, meta-messages, and an object-oriented approach to describe the semantics of our language
- defining a translation of the object-oriented database requests into a relational language

The paper is organized as follows. In section 2 we describe the object-oriented schema, and provide its graphical and symbolic representations using an example database. In section 3 we define a subset of our language to operate on the underlying database. Section 4 discusses the translation of object-oriented queries into extended SQL expressions. A summary is given in section 5.
2. MODEL DESCRIPTION

In order to integrate a database system with an object-oriented programming system, it is necessary to develop a data model that would be compatible with the object-oriented language. In this section we shall present an object-oriented data model that is fully integrated with Smalltalk.

The basic concepts of the proposed data model are generally similar to those of other object-oriented database systems, but significantly different in the sense that messages are usually sent to instances of classes that are sets of objects. These concepts are shortly described as follows:

- objects, which are the most elementary components of the conceptual view of the modeled world.
- object type, which in the usual way allows objects to be classified.
- object sets, which are objects of the same type.
- maximal object set, which contains all objects of a given type.
- class, which contains subsets of the maximal object set for some object type.
- class instance, one of the subsets for the class
- minimal object set, which contains one object from the current class instance.
- starred object set, which is one of the subsets of the current class instance.
- messages, that are sent to a class instance to perform some action that usually results in an instance of another class.
- receiver, for a given message, which is the class instance to which the message is sent.

Our object-oriented database consists of classes with associated messages. Messages are usually sent to a set of objects constituting an instance of a class (which can be sometimes a one-element set or the empty set). One class may be a subclass of another, in which case it inherits all the messages associated with the superclass.

Among the messages defined for our language, there are special messages which we call database messages. These messages are defined on the underlying database, and cannot be modified by users. Our approach allows for representing all types of relationship by messages. Therefore each database message can be inverted by applying a higher-order message 'inverse'. A higher-order message is similar to higher-order functions (e.g., in LISP) in the sense that it accepts another message as argument. Another example of a higher-order message is 'rename', that can be used to change the name of another message. There are also binary messages that allow operations on more than one class, e.g., union and intersection. Additionally, it is possible to send aggregate messages (COUNT, SUM, AVERAGE, etc.) to certain classes. We allow typical Smalltalk keyword messages for simple selections. Among keyword messages we have block messages, to encapsulate a sequence of actions and execute it in a different context. Although throughout the paper we are using the conventional term 'set', we more precisely mean a Smalltalk 'bag' that allows duplicates. Therefore it is necessary to include in the language a new message 'distinct', that removes all duplicates. Users may also define their own messages for a given class.

As an example, let us consider the Road database, adapted from [10] and whose object-oriented schema is shown in figure 1. In the schema, and throughout this paper, the class names are always in upper-case: maximal object sets start with an upper-case letter, followed by a sequence of lower-case letters; message names and minimal object set names are all in lower-case. The object-oriented schema contains, among others, object classes ROAD, NAME and TYPE. The maximal object set for class ROAD contains all existing roads. The possible instances of the class ROAD are the subsets of the maximal object set. Initially, the class ROAD contains only one instance (the maximal object set) but other instances may be created dynamically during the execution of object-oriented programs. An instance of a class can be given explicitly by enclosing the objects within parentheses and preceding with a pound (#) sign. The messages are specified on the diagram by means of arrows from the receiver class to the result class. For example, is_type_of is a message sent to an instance of the class TYPE, which would result in the creation of an instance of the class ROAD. For example, if we have an instance #Highway of the class TYPE, sending the message is_type_of to this class instance would result in a set of identifiers of roads, for those roads whose type is 'Highway'. If we then send the message has_name to that set, the result would be a set of names such as #(I-45, I-59). It can be seen from figure 1 that each road comprises several components, each of which is either a road segment or an intersection. Therefore we model the classes ROAD_SEGMENT and INTERSECTION as subclasses of COMPONENT.

The schema in figure 1 contains only the initial database messages. In general, it can also include user-defined
Figure 1 The object-oriented schema for 'Road' database
3. OBJECT-ORIENTED DATABASE LANGUAGE

The basic constructs of our language are Smalltalk classes, instances of classes, and messages. The language requires several extensions to Smalltalk, however, in order to allow for database operations. The syntax of some of these extensions will be provided in this section (for a more complete description, see [3]).

To simplify the presentation, we shall present an abstract syntax for the database requests. The precedence rules follow the usual Smalltalk approach, that unary messages have higher precedence than binary messages and keyword messages have lower precedence than either. We first consider queries involving database messages and then describe additional features of the language such as simple selection messages, aggregate messages, binary messages, block messages, user-defined messages and inheritance.

3.1 Database messages, selection messages and aggregate messages

A typical query would consist of a class instance name corresponding to the maximal object set for that class, followed by a sequence of database messages.

\[ \text{<simple-query>} ::= \text{<class-instance-name>} \]

\[ \text{<simple-query>} ::= \text{<simple-query>} \text{ <db-message>} \]

A database message is a message defined in the underlying database. A simple query can also contain selection messages, which specify a subset of the current object set. Therefore the syntax of simple query can also be defined as:

\[ \text{<simple-query>} ::= \text{<simple-query>} \text{ <simple-selection-message>} \]

A simple selection message is a keyword message (EQ, LT, GT, NE, etc.) that requires an argument that can be a constant or a simple query. GT and LT require one-element sets as the argument, and EQ and NE can accept any set as argument. An example of a valid simple query containing a simple selection message is \( \text{Name EQ: #(Main)} \) \text{is-name-of has-type} , which specifies the following query:

\[ \text{Query 1} \]

"Give the type of the road named 'Main'"

The first message, EQ, is sent to the maximal object set for the class NAME. This message is a keyword message, and therefore requires an argument which in this case is a one-element set containing 'Main'. The result of this operation is an instance of the class NAME that is receiver of the next message "is-name-of". The last message in the above query is "has-type", that is sent to the result of the message "is-name-of". The application of the message "has-type" produces the requested set of types for the roads named 'Main'.

Additionally, a simple query can contain an aggregate message, as defined below:

\[ \text{<simple-query>} ::= \text{<simple-query>} \text{ <aggregate-message>} \]

Aggregate messages perform operations such as COUNT, SUM, AVERAGE, MIN, MAX, etc., on the current object set to yield a set containing a single (aggregate) numerical value.

Queries discussed in this sub-section are a subclass of general requests, as is shown by the BNF statement below:

\[ \text{<request>} ::= \text{<simple-query>} \]

3.2 Queries involving binary messages

We allow the usual set operations, i.e., UNION, INTERSECT and DIFFERENCE. The syntax of queries involving such operators is as follows:

\[ \text{<simple-query>} ::= \text{<simple-query> \text{ <set-operator> <simple-query>''}} \]

A query may consist of several simpler queries linked by set operators. The order of application of the set operators is made explicit by the use of parentheses. These set operations can be treated as binary messages.

3.3 Queries involving block messages

The above-described messages are applied to the whole set of values for the current class instance. In order to perform grouping and aggregation operations, computations and complex selections, it is necessary to apply a
sequence of messages to each element of a set individually. This can be performed by sending a block message that requires a Smalltalk-like block as an argument. The first block message, called group_by, allows for grouping and aggregation operations. Its syntax is as follows:

\[
\text{<simple-query>} ::= \text{<simple-query>' group_by: [ <extended-simple-query> ]}
\]

The extended simple query is very similar to a simple query, except that the receivers of extended-simple-query can also be minimal object sets belonging to a class. These minimal object sets are specified by the class name in lower case. The restriction is that the class of such sets should be the result class of simple-query'. For example, a valid simple-query is Road group_by: [ road contains segment_spec has_speed_limit average ] , which specifies the following:

\[Query 2\]

"For each road, give the average speed limit over all its road segments"

There are some other situations where we need to apply operations to each element of the current object set individually. The 'compute' block message allows us to specify arithmetic operations to derive new values for each element of the current set. The syntax is as follows:

\[
\text{<simple-query>} ::= \text{<simple-query>' compute: [ <arithmetic-query-expression> ]}
\]

where the arithmetic query expression contains extended simple queries, constant sets and arithmetic operators such as +, -, *, /. These operators are defined on one-element sets.

Also, it is helpful to select elements from the current object set using complex selection predicates similar to the approach of [9]. The 'select' block message would allow such selections. The syntax is as follows:

\[
\text{<simple-query>} ::= \text{<simple-query>' select: [ <boolean-query-expression> ]}
\]

where the boolean query expression contains extended simple queries or arithmetic query expressions, and boolean-valued operators such as =, >, <, <=, IN, etc. These operators are defined on sets. We assume that >, < are defined on one-element sets, and =, <= is defined on general sets. Boolean query expressions can also be constructed using simpler boolean query expressions and logical connectives such as AND, OR, NOT.

3.4 Defining new messages

The result of any simple query is a set of objects. A simple query can thus be viewed as a compound message to compute such objects. To facilitate the expression of subsequent queries, it is often convenient to create a new message corresponding to the query. This task is performed by the message 'define_message', which is sent to the instance of class DATABASE that contains information about classes and messages. The syntax of message definitions is:

\[
\text{<message-definition>} ::= \text{database define-message: <message_name> for: <class_name> as: [<starred-simple-query>]}
\]

where starred simple query is similar to a simple query, except that the initial receiver is the starred class name. In case of set operations, for the last-performed set operation, at least one argument should have starred class name as the first receiver.

When the operator define-message is invoked, the new message is defined for each member of the class, according to the starred simple query.

3.5 Inheritance

A subclass automatically inherits the messages of its superclass, in the usual way for object-oriented systems. For example, the class ROAD_SEGMENT inherits the messages of the class COMPONENT. Inheritance of the message 'is_part_of' provides a direct link between the ROAD and ROAD_SEGMENT classes, thereby simplifying query formulation. For example, suppose we want to find the names of all roads which have at least one road segment with a speed limit of 55 mph. This query can be written as (Speed-limit EQ: #(55)) is_limit_of is_part_of has_name .

\[Query 3\]

"Give the names of all roads having at least one segment with speed limit 55"

In case of multiple inheritance, it might happen that two superclasses have messages of the same name. We resolve this kind of conflict by explicitly specifying the superclass in the object-oriented query. In such a case,
query 3 should be expressed as (Speed_limit EQ: #(W) is_limit_of is_a is_gart_of has_name).

We also have an interesting case of automatic selection of subclasses, which is related to inheritance. This happens when we want to select a subclass from the current class, and apply a message to it. An example is query 2 defined above. This query could be rephrased as Road_group_by:[ road contains has_speed_limit average ].

The message segment_spec was unnecessary because there is no ambiguity in choosing the receiver for has_speed_limit.

4. TRANSLATING QUERIES INTO DATABASE LANGUAGES

One of the major tasks performed by the integration module is the translation of identified database requests into expressions in a language supported by the DBMS. In this section, we shall discuss the translation of these database requests into an extended version of SQL which supports features such as composite attributes, general arithmetic expressions in the SELECT clause, etc. To facilitate such translation, we specify semantic rules for each of the BNF statements defining the grammar.

To translate database requests into a relational language, it is necessary to define a mapping of the object-oriented schema to a relational schema. Here we are using a simple mapping of the object-oriented schema into binary relations, where each relation corresponds to a message. To allow for the same message name being used by different messages (with different receivers) we concatenate the receiver name with the message name to form a relation name, with a hyphen between. The attribute names are the receiver class and the result class. For example, some relations for the object-oriented schema shown in figure 1 are listed below:

ROAD-HAS_NAME (road, name)
ROAD-CONTAINS (road, component)
SHAPE-IS_SHAPE_OF (shape, road_segment)
ROAD_SEGMENT-HAS_LENGTH (road_segment, length)

These relations should not be considered as stored relations, but rather as views defined on a database designed in a standard way. The same applies to views corresponding to the maximal object sets, e.g.,

ROAD (road)
ROAD_SEGMENT (road_segment)

Based on this mapping, we shall define the translation of database requests into relational expressions. We shall express the translation rules using an object-oriented approach. In order to do that, we need to define a meta-schema that would describe meta-data, i.e., information about all classes and associated messages. The diagram representing part of the meta-schema is shown in figure 2. The diagram contains meta-objects such as classes, class names, messages, message names. It also contains meta-messages such as has_name, is_name_of, has_message and is_message_of. The meta-schema also includes meta-messages not explicitly shown in the diagram. These are UNION, INTERSECTION, etc. We also use the standard Smalltalk notation for appending strings (a comma). Symbols (constant strings) are denoted by a pound sign (#) followed by a sequence of characters. For each BNF statement, we shall specify translation rules (semantic actions). We shall assume that each non-terminal will have certain variables (semantic attributes) in terms of which we can express queries. These variables are:

(i) result-class - class of the object set returned after evaluating all components
(ii) relation - a relation in the underlying database, corresponding to the query.
(iii) attribute - an attribute of the relation described in (ii), corresponding to the result class defined in (i)
(iv) SELECT, FROM, WHERE, GROUP BY - the partial SQL clauses
(v) expression - a complete SQL statement (a SQL view, or query, or both)

In describing the translation rules, we shall follow the same sequence as in section 3.

4.1 Translation of database messages, simple selection messages and aggregate messages

The most elementary form of query, involving only a maximal object set, can be described as follows:
The translation rules associated with this BNF statement are trivial, and simply assign the values of variables of class-instance-name to corresponding variables of simple-query. Next we present the translation rules for a simple query including database messages.

\[ \text{<simple-query> ::= <class-instance-name>} \]

The SELECT clause contains only the value of the attribute variable, that is constructed by appending the name of the relation corresponding to the relation variable with the result class. To the FROM clause of the simple-query', we add (using UNION) the name of the relation corresponding to the database message. To the WHERE clause we add the join condition for joining the relation corresponding to the current database message with the relation corresponding to the simple-query'. For this reason, the relation variable should be maintained in each recursion step, together with the attribute and the result class.

The complete SQL query is obtained by appending the SELECT, FROM and WHERE clauses of the simple query.

\[ \text{<request> ::= <simple-query>} \]

To support queries with simple selection, we define the translation rules as follows:

\[ \text{<simple-query> ::= <simple-query'> <selection-message>} \]

\[ \text{request.result-class <= simple-query.result-class} \]
\[ \text{request.expression <= simple-query.expression} \]
\[ \text{#SELECT , simple-query.SELECT} \]
\[ \text{#FROM , simple-query.FROM} \]
\[ \text{#WHERE , simple-query.WHERE} \]

In this case, we append to the WHERE clause the selection condition. The selection message expression consists of the comparison operator and the constant or derived relation. For example, query 1 is translated to the following SQL expressions:

\[ \text{Query 1} \]
\[ \text{SELECT ROAD-HAS-TYPE.type} \]
\[ \text{FROM NAME,ROAD-HAS-TYPE,N-NAME-IS-NAME_OF} \]
\[ \text{WHERE NAME.name = 'Main'} \]
\[ \text{AND NAME.name = NAME-IS-NAME_OF.name} \]
\[ \text{AND NAME-IS-NAME_OF.road = ROAD-HAS-TYPE.road} \]

This allows us to specify simple queries without aggregate functions. We next need to consider aggregate queries.

\[ \text{<simple-query> ::= <simple-query'>} \]
\[ \text{<aggregate-message>} \]
simple-query.result-class <= simple-query'.result-class

simple-query.relation <= generate-new-view

simple-query.expression <= simple-query'.expression, #CREATE, #VIEW, simple-query'.name, #, #AS, #SELECT aggregate-message.expression, #, simple-query'.attribute, #, #FROM, simple-query'.FROM, #WHERE, simple-query'.WHERE

simple-query.attribute <= simple-query'.attribute, #

simple-query.SELECT <= simple-query.attribute

simple-query.FROM <= simple-query.relation

In this definition we assumed that, after applying aggregate operations to a set of objects belonging to a given class, the result belongs to the same class. However, for the COUNT aggregate operation, the semantic definition should be modified, for example, by defining the resulting class as the class of numbers.

The above translation rules can be modified to simplify SQL expressions. For example, when the aggregate message is the last message in a query, it is not necessary to create a view.

4.2 Translation of binary messages

When a binary message is specified, it is necessary to translate each of its arguments into a complete SQL expression, and to define a view involving a set operation corresponding to the binary message. These complete SQL expressions are formed by simply appending the SELECT, FROM and WHERE clauses, as shown below.

\[
\text{<simple-query>} ::= \text{<simple-query>' <set-operator> <simple-query>}'
\]

simple-query.result-class <= simple-query'.result-class /*simple-query'.result-class = simple-query''.result-class*/

simple-query.relation <= generate-new-view

simple-query.expression <= simple-query'.expression, simple-query''.expression, #CREATE, #VIEW, simple-query.relation, #, simple-query.result-class, #, #AS, #
As with the aggregate queries, the creation of a view can be avoided if the set operation is the last operation in the query.

4.3 Translation of block messages

Group-by query is defined by the following BNF statement:

\[
\text{<simple-query>} \::= \text{<simple-query'> group-by: [<extended-simple-query>]}
\]

The semantics of the translation is very complex in the general case. In case the simple-query' is a maximal object set, and extended-simple-query does not contain any set operations, the translation is very similar to the previously-defined rules. Additionally, a GROUP BY clause is created and contains the attribute corresponding to the class name of the receiver of the group-by message. For example, query 2 will be translated as follows:

**Query 2**

```
SELECT AVG (ROAD_SEGMENT-HAS-SPEED_LIMIT.speed_limit) 
FROM ROAD, ROAD-CONTAINS, COMPONENT-COMPONENT-SEGMENT-SPEC, ROAD_SEGMENT-HAS-SPEED_LIMIT 
WHERE ROAD.road = ROAD-CONTAINS.road 
AND ROAD-CONTAINS.component = COMPONENT-COMPONENT-SEGMENT-SPEC.component 
AND COMPONENT-COMPONENT-SEGMENT-SPEC.road_segment = ROAD_SEGMENT-HAS-SPEED_LIMIT.road_segment 
GROUP BY ROAD-CONTAINS.road 
```

In case the simple-query' contains messages, and extended-simple-query does not contain any set operations, the translation is performed in two steps. First the view corresponding to simple-query' is created, then the view is used in translation of the extended simple query. In the general case, when binary messages are present in the extended simple query, the translation process should be extended so that each view created will contain an additional attribute corresponding to the class of the receiver of the group_by message. If the group_by message is not the last message in the query, the corresponding view should be created.

Queries involving computations are specified by the following syntax:

```
<simple-query> ::= <simple-query'> compute: [<arithmetic-query-expression>]
```

If simple-query' is a maximal object set, and each extended simple query in the arithmetic query expression can be translated without creating a view, then the translation would involve the union of the FROM and WHERE clauses of the extended simple queries. The SELECT clause would be constructed from the arithmetic-query-expression by substituting an attribute for each extended simple query.

In the general case, when for example binary messages are present in the extended simple queries, the translation process should be extended so that each view created will contain an additional attribute corresponding to the class of the receiver of the compute message. If the compute message is not the last message in the query, the corresponding view should be created.

Queries involving complex selections are specified by the following syntax:

```
<simple-query> ::= <simple-query'> select: [<boolean-query-expression>]
```

This is translated by appending to the WHERE clause of the simple-query' an extra condition involving an IN operator and a subquery which corresponds to the boolean query expression. For a more complex boolean expression, for example including binary messages, the translation process should be extended so that each view created for the boolean query expression will contain an additional attribute corresponding to the class of the receiver of the select message.

4.4 Translation of new message definitions

Message definitions should result in creation of a relational view that can be later used in a query.

```
<define-message> ::= database define_message: <message-name> for: <class-
```
The starred simple query is translated into SELECT, FROM and WHERE clauses, similarly to the simple query. However, if during this translation a new view needs to be created, such views should contain an additional attribute corresponding to the class name.

In addition, the meta-schema is modified to include the new message in the set of messages, and the receiver and resulting object types are associated with the new message.

4.5 Inheritance

When we use a message which is inherited from some superclass, the translation process is very similar to translation of queries without using inheritance. In this case there is no message applied directly to the subclass, so we need to use the superclass name in specifying the relation. It can be accommodated by modifying the translation rules for all productions containing database message on the right-hand side. The current object type should be substituted by the appropriate superclass type, which can be determined by a meta-message search_for_superclass_with. For example, query 3 is translated as follows:

Query 3

```
SELECT ROAD-HAS-NAME.name
FROM SPEED_LIMIT, SPEED_LIMIT-IS-LIMIT-OF.
    COMPONENT-IS-PART-OF, ROAD-HAS_NAME
WHERE SPEED_LIMIT.speed_limit = 55
AND SPEED_LIMIT.speed_limit = SPEED_LIMIT-IS-LIMIT-OF.speed_limit
AND SPEED_LIMIT-IS-LIMIT-OF.road_segment =
    COMPONENT-IS-PART-OF.component
AND COMPONENT-IS-PART-OF.road =
    ROAD-HAS-NAME.road
```

Automatic selection of subclasses, discussed in section 3.5, can be handled in a very similar way. If we want to implement both inheritance and automatic selection of subclasses, we need to search both the superclasses and the subclasses for the specified message. Assuming that there is no ambiguity, the current object type should be substituted by the appropriate superclass or subclass type.

5. SUMMARY

We have described an object-oriented data model, that allows uniform specification of database requests and application programs. Using this model, database requests can be represented by messages sent to instances of database classes. We introduced various kinds of messages that make query specification more natural for the Smalltalk programmer. We provided the semantics of these messages corresponding to database requests, in the form of rules for translation into a relational language. The rules are specified using a meta-model, meta-messages and an object-oriented approach.

REFERENCES