Query Translation Supporting the Migration of Legacy Databases into Cooperative Information Systems

Research

Daniel A. Keim, Hans-Peter Kriegel, Andreas Miethsam
Institute for Computer Science, University of Munich
Leopoldstr. 11 B, D-80802 Munich, Germany
e-mail:{keim, kriegel, miethsam}@informatik.uni-muenchen.de

Abstract

In this paper, we present a query translation algorithm which allows object-oriented queries to be automatically translated into a relational query language. Our goal is to provide a unique and powerful query interface supporting the cooperation of information systems, particularly under the aspect of migrating existing systems without changes into the cooperation. Translation algorithms, like the one proposed in this paper are essential parts of CISs to mediate between the common global query language and the query languages of participating components. Our query translation algorithm ensures a fully automatic translation of object-oriented queries into equivalent SQL queries for the original relational schema in all cases where a direct translation is possible. In all other cases, it generates SQL queries providing a superset of the desired data and a sequence of 'formatting' functions that transform the data into the desired result.

1. Introduction

Nowadays, many private, commercial and research information services are publicly accessible over wide area networks ranging from simple file distribution, e.g. via 'anonymous ftp', to information systems (IS) with more complex query interfaces, e.g. 'mosaic' or library systems with text retrieval facilities. However, often people do not know where to find relevant data, how to retrieve data, or even do not know about the existence of a system useful for solving their information needs. On the other hand, even within one company or institute, numerous heterogeneous information systems with various data models, query languages and interfaces are used in research and industry to perform different tasks or simply for historic, market or individual preference reasons. Many tasks, however, require access to more than one of these information systems at a time. For example, in a hospital, the treating doctor needs information from previous examinations stored in various systems, e.g. an image from the X-ray image database, blood data from the system of the pathology laboratory and the daily record of patient data from the wards. To present another example, several institutes record various types of environmental data; it is very desirable to combine these flat data or to combine them with maps of roads, industry, etc. provided by GISs to detect and evaluate complex interactions.

The need to integrate or transparently access different systems has been realized by many people. The naive solution of integrating all systems into one fails in most cases for several reasons. It usually requires a complete system change or migration making it necessary to convert the existing databases including all their application programs that have been successfully used over the years. A further difficulty in the migration process is that, in general, the systems are used on-line with many application programs running permanently on a daily basis. In performing a system change or migration, most companies fear the possible loss of data and the necessary changes of application programs. Additionally, because of the diversity of tasks, one system often is not capable of efficiently performing all of them. Finally, the number and the autonomy of systems which are accessible via worldwide networks prevents the desired kind of integration. Among the current solutions are specifically designed application programs which access relevant systems in their own languages and combine the results before further processing. Also, most database vendors offer gateways that provide some kind of cross-database access [Syb 90, Ing 92, Ora 92, Inf 92] allowing the use of specific new database systems in conjunction with existing relational ones. Another solution is data exchange by flat files exported from one system and imported into a target system. These solutions, however, have the drawbacks of high development and maintenance costs, as well as the lack of flexibility. Furthermore, the resulting systems, i.e. the applications, are even more proprietary than the integrated systems.

The common goal of research activities in the fields of database systems, (intelligent) cooperative information systems (CIS), interoperable systems, multidatabase systems is the development of general concepts supporting the migration or integration of previously isolated systems into CISs, e.g. [BHP 92, EJ 92, RPR 89, SL 90]. According to [BC 92], we define

- an IS to be any system that allows shared access to persistent data,
- a CIS to be an arbitrary number of component ISs, that interact in some way to execute joint tasks.

Particularly, the components must be able to interoperate directly or indirectly, i.e. to mutually send, receive and understand(!) requests and results. We additionally assume that

- the communication language is high level, e.g. an object-oriented query language,
- information on the components (schema, location) is globally available,
- the component ISs are at least able to receive requests and send back answers in a client server fashion, i.e. play a passive role in the cooperation.
- there exists at least one active component in the CIS. Active components are able to send, forward, distribute requests and to combine results,
- existing ISs need not to be changed in order to be integrated into the CIS, that means access to these systems uses already existing query interfaces.

Figure 1 shows a conceptual CIS architecture meeting the above requirements. Especially important in our opinion is that existing (legacy) systems are integrated into the CIS by viewing them through a 'cooperative interface'. In this case, no change to the legacy systems is necessary and previous applications may be preserved. Therefore, cooperative interfaces help to solve part of the overall migration problem which, in our notion, is a summation of problems on several layers. These comprise among others how to establish a communication between different networks, hardware platforms, and operating systems, add client server facilities to components, provide schema transformation and query translation modules, deal with the autonomy of the components (in the integration as well as the operation phase). In dealing with legacy systems which are heterogeneous with respect to their data models and query languages, it is especially important to transform their schemas into the global data model and make them globally available. Furthermore, the cooperative interfaces must be able to unify languages and models in addition to controlling communication and

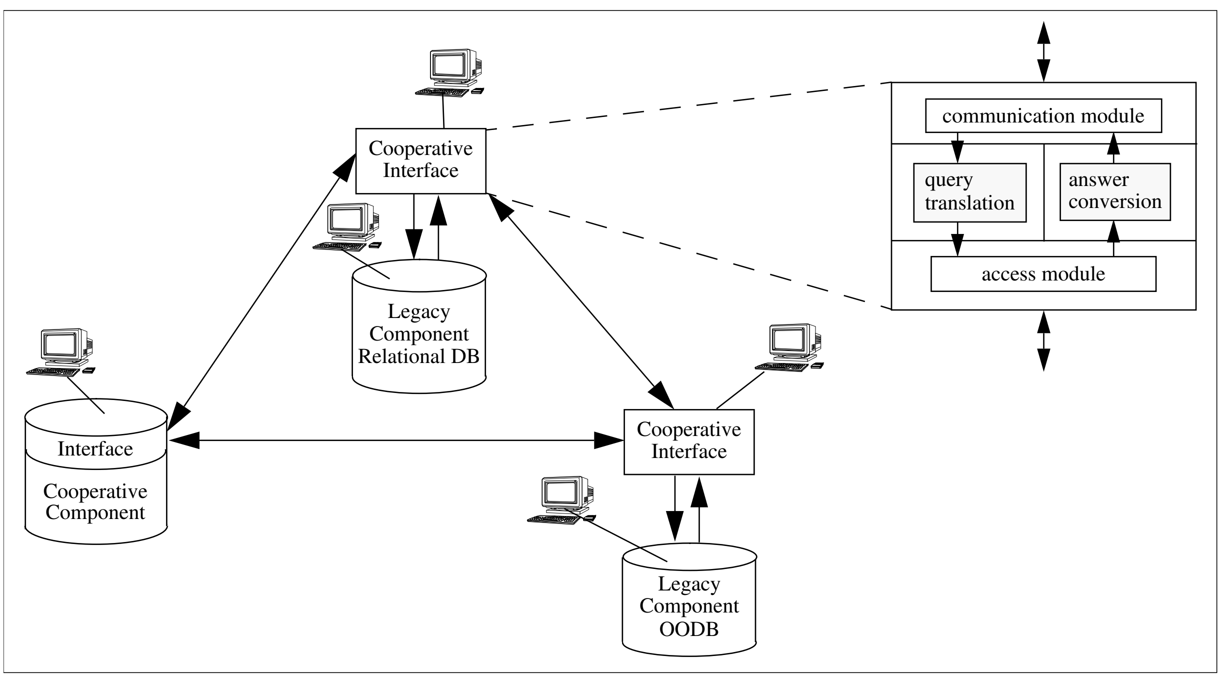

Figure 1: Architectural Concept

access. This means, for the cooperation of ISs modules are needed which translate requests in the global language on objects represented in the global model into requests in the local language and reformat results into the global language. For these tasks, the cooperative interfaces must access the global information to map between local and global object representations. Further submodules of the cooperative interfaces not listed in figure 1 comprise query decomposition, result combination, optimization and global transaction monitoring.

As a core technology for the integration of legacy ISs into a CIS, in this paper we focus on the query translation and answer conversion, specifically from an object-oriented language to the relational SQL. The object-oriented data model and language seem to be well suited as global language for communicating ISs because of its message passing paradigm, because of its rich semantics which allow representation of most other models as well as systems with operational interfaces, and because of its availability in the form of object-oriented databases. We choose SQL as translation target language, since many instances of relational systems are currently installed and accessible using a standard SQL [ISO 92] interface. To apply this approach to a real world environment with isolated, heterogeneous ISs, additional translation modules have to be developed to adapt the cooperative interface to non-relational components.

The rest of the paper is organized as follows: Section 2 introduces the overall framework and gives a brief overview of the schema enrichment and transformation as well as a short introduction of our Structured Object Query Language (SOQL) which provides declarative query facilities for objects. In section 3, we then present the steps that are necessary in automatically translating SOQL queries for the created object-oriented schema into equivalent SQL queries for the original relational schema. In section 4, we describe the formatting process that is needed to transform the flat results provided by the relational system into structured results that are specified by the object-oriented query. Section 5 summarizes our approach, points out some problems and gives directions of future research.

2. The Framework

In the following, we are going to briefly introduce two prerequisites of our query translation algorithm, namely the schema enrichment and transformation algorithm on the one hand and the Structured Object Query Language (SOQL) on the other hand.

2.1 Schema Enrichment and Transformation

Since, in general, object-oriented schemas contain more semantics than corresponding relational schemas, more input than the pure relational schema is needed to produce adequate, well-structured object-oriented class definitions. The needed additional semantic information includes information on tables representing relationships, the type of the relationship (1:1, 1:n, n:m), attributes or groups of attributes representing foreign keys and so on. This information may either be provided by the database administrator or, in some cases, it may be deduced from an underlying entity-relationship design schema. It is stored as part of the meta information which includes all information on the enriched relational schema, on the created object-oriented schema and on the mappings between them. As we will see in the next sections, the meta information is crucial not only for the schema transformation process but also for an automatic translation of SOQL queries.

The basic steps of the schema transformation algorithm are as follows. First, each relation is translated into a class definition with each relational attribute becoming a member variable. Next, all functional relationships are replaced by direct object references, in one direction by a simple object reference, in the other direction by a set-valued object reference. All remaining n-ary relationships are translated into methods with one method providing the set of tuples that fulfill the relationship and one method for each relationship attribute. The additional methods are added to each class that is part of the relationship. In figure 2, an example for a relational database Flight-DB together with the corresponding object-oriented schema is given. The details of the schema transformation algorithm are beyond the scope of this paper. A formal description can be found in [KKM 93a].

At this point, it should be mentioned that the schema created by our schema transformation algorithm may not provide a perfect object-oriented schema. It does not use all object-oriented modeling features (e.g. subtyping, inheritance) but it still provides a semantically enriched, well-structured object-oriented schema that allows SOQL queries to be significantly shorter and more intuitive than corresponding SQL queries using the original tables. Let us further emphasize that only object-oriented class definitions are generated with the instances remaining in the relational database. Thus, access operations to instances of object-oriented classes have to be translated into accesses to the corresponding relational tuples which is done by our query translation algorithm (c.f. section 3).

FlightDB:

Passenger (<u>pid</u>: Integer; name: String; address: String)
Departure (<u>did</u>: Integer; start: Date; flight: Integer; airline-id: String; plane-id: Integer)
Pass_Dept (<u>did</u>: Integer; <u>pid</u>: Integer; booking: Date)
Airline(<u>airline-id</u>: String; name: String)
Plane(<u>serial-nr</u>: Integer; ...) . . .

```
Class Passenger with
  attributes
    pid: Integer;
    name: String;
    address: String;  key is (pid);
  methods
    departures: → Set (Departure);
    booking: Departure → Date;
end;

Class Airline with
  attributes
    airline-id: String;
    name: String;  key is (airline-id);
  methods
    departures: → Set (Departure);
end;

Class Departure with
  attributes
    did: Integer;
    start: Date;
    flight: Integer;  key is (did);
  methods
    airline: → Airline;
    plane: → Plane;
    passengers: → Set (Passenger);
    booking: Passenger → Date;
end;

Class Plane with
  attributes
    serial-nr: Integer;
    ...
end;    . . .
```

Figure 2: Example for the Schema Transformation

2.2 Structured Object Query Language

In this subsection, we give a short introduction to our Structured Object Query Language (SOQL). SOQL is a declarative query language for querying the created object-oriented schema. It is an easy-to-use but powerful and orthogonal extension of SQL. It is similar to other declarative query languages for object-oriented database systems (O_2SQL [BCD 92], Object SQL [HD 91], OSQL [Fis 89], OQL [ASL 89]) but provides additional features such as the generalization of the dot-notation and structured expressions. The basic query format of SOQL can be indicated by the following description

select
  {<range_var>{.<method>}*{.struct_expr}$^{0/1}$ }$^{+}$
for each
  {<classname>{.<method>}* <range_var>}$^{+}$
{ where <condition> }$^{0/1}$.

According to the expression in the 'select' clause, a new (temporary) object class is automatically created with all tuples fulfilling the condition being available as virtual instances of this class. The result is also available as a (nested) set and can therefore be directly used in nested queries. As indicated in the query format definition, methods are applied to class or range variables using dot-notation. Chains of methods may be connected in dot-notation as long as the methods are defined for the corresponding class. The chaining of methods allow direct access of one object class from another without explicitly joining them. It is a form of schema navigation in the created object-oriented schema. In the condition, all methods including the created access methods to attributes may be used as long as the result of the whole expression is of result type 'Boolean'. Special features of SOQL are structured expressions and the generalization of the dot-notation. Structured expressions allow an easier specification of queries with structured results by providing the possibility of defining the result structure by square brackets. The generalization of the dot-notation to sets is an intuitive but powerful extension of the normal dot-notation (c.f. section 3). To provide the basic queries facilities that are available in SQL, a set of basic object classes (*Boolean, String, Numbers, Integer, Real* and the generic classes *Set* and *List*) together with a set of basic methods including the aggregate operations *count, avg, sum*, *min*, *max* (Set(Numbers) → Numbers) is predefined. A detailed description of SOQL can be found in [KKM 93b].

To further illustrate our query language, in the following we will give two examples for SOQL queries. For the query examples, we use the transformed example database as presented in figure 2. A simple query selecting all passengers and their addresses that fly with airline ‘Lufthansa’ on the ‘06/18/93’ would be expressed as

Example 1:

```
select   P.name, P.address
for each Passenger P, P.departures D
where    D.start = '06/18/93' and
         D.airline.name = 'Lufthansa'
```

In the second query example, all passengers, their addresses and flights with flight numbers, list of passengers for each of the flights and total number of flights for each passenger are selected for all passengers which have addresses containing ‘80802 München’.

Example 2:

```
select   P.[name, address], P.departures.
              [[did, passengers.name], count]
for each Passenger P
where    P.address like '%80802 München%'
```

The query examples will be used in sections 3 and 4 to explain the query translation algorithm. Note, that the result of the second query is of the complex type

Set ([String, String],
[Set ([Integer, Set (String)]), Integer]).

Nested results may occur as answer for queries with structured expressions or queries where the generalization of the dot-notation is used more than once in a row. Furthermore, in corresponding SQL queries additional information is needed to do the grouping and aggregation (e.g. the counting of departures) which is only implicit in the SOQL query. In general, if the result for a query is a nested set with more than one nesting level, there is no one-to-one translation to an SQL query. Equivalent SQL queries for our query examples are given as results of the query translation algorithm in section 3.

To sum up, SOQL provides query facilities that allow queries to be much shorter, easier to write and understand and more intuitive than corresponding SQL queries. Since the created class definitions are more structured, in most cases joins do not have to be specified explicitly and complex queries are avoided. In addition, the results of SOQL queries can be arbitrarily structured and the application of methods in dot-notation is generalized to work on sets.

3. Translation of SOQL Queries into SQL-Queries

Since information is added during the schema transformation process and SOQL has more expressive power than SQL, it is obvious that all queries expressed in SQL over the relational schema (RS) can also be expressed by SOQL queries over the created object-oriented schema (OS). This section deals with the translation of SOQL queries into standard SQL [ISO 92] and the identification of formatting primitives during the translation process which are needed to restructure the result according to the complex answer type given by the SOQL ‘select’ clause. To illustrate the tasks of the translation algorithm, figure 3 shows an example for a small relational database and the virtual instances of the corresponding object-oriented schema. The virtual instances of the object-oriented database ODB are created from the tuples of the relational database RDB by a *virtual instance mapping* v_{inst}: (OS, RDB) $\mapsto$ ODB. By the virtual instance mapping, basically, each tuple of a non-relationship table of RDB is mapped to a virtual instance of the respective class in ODB and each tuple or attribute representing a relationship is mapped to a virtual object reference. The basic idea of our instance mapping is similar to the one presented in [Heu 89] which has been proposed to formally describe schema equivalence of a semantic, a nested relational and a relational data model. Executing the SOQL query in figure 3 against ODB yields the structured result {(Smith, {401, 403}), (Smith, {401})}. A corresponding SQL query together with its result is also given in figure 3. Although both query results seem to be very similar, it is impossible to create the structured SOQL result from the flat result of the SQL query if no additional information is available. However, by adding the key attribute P.pid of Passenger to the SQL ‘select’ clause, we get the result {(Smith, 1, 401), (Smith, 1, 403), (Smith, 2, 401)} which can easily be transformed into the desired format by grouping the tuples according to P.pid, combining the D.did attributes to sets and afterwards projecting out the P.pid attribute. Selecting additional information that allows to structure the results from the relational database into the desired format, is one of the ideas which is used in our translation algorithm. The main tasks of the translation algorithm are

- resolving chains of method applications by suitable joins and subqueries on the relational side,
- flattening the nested structure while simultaneously creating the inverse formatting operations,
- correctly replacing the SOQL condition part by equivalent SQL constructs which may involve handling of methods on structured types, set operations and so on. However, for the presentation of the translation algorithm in subsection 3.2 we restrict ourselves to SQL-like conditions and discuss feasible extensions separately in subsection 3.3.

RDB:

Passenger

pid	name	address ...
1	Smith	New York. ...
2	Smith	London ...
3	Jones	Paris ...
4	Huber	München ...

Departure

did	start	flight	...
401	7-1-93	0815	...
402	7-1-93	1414	...
403	7-1-93	1017	...

Pass_Dept

pid	did	booking
1	401	...
1	403	...
2	401	...
4	401	...
4	402	...

(virtual) ODB:

Passenger = $\{o_1, o_2, o_3, o_4\}$

o_1.pid = 1, o_1.name = 'Smith', o_1.address = 'New York ...', o_1.departures = $\{o_5, o_7\}$
o_2.pid = 2, o_2.name = 'Smith', o_2.address = 'London ...', o_2.departures = $\{o_6\}$
o_3.pid = 3, o_3.name = 'Jones', o_3.address = 'Paris ...', o_3.departures = $\{\ \}$
o_4.pid = 4, o_4.name = 'Huber', o_4.address = 'München ...', o_4.departures = $\{o_5, o_6\}$

Departure = $\{o_5, o_6, o_7\}$

o_5.did = 401, o_5.start = '7-1-93', o_5.flight = '0815' o_5.passengers = $\{o_1, o_2, o_4\}$
o_6.did = 402, o_6.start = '7-1-93', o_6.flight = '1414' o_6.passengers = $\{o_4\}$
o_7.did = 403, o_7.start = '7-1-93', o_7.flight = '1017' o_7.passengers = $\{o_1\}$

SOQL:

```
select  P.name, P.departures.did
for each  Passenger P
where  P.name='Smith'
```

result:

{(Smith, {401, 403}), (Smith, {401})}

SQL:

```
select  P.name, D.did
from   Passenger P, Departure D, Pass_Dept Pd
where  P.name='Smith' and
         P.pid=Pd.pid and Pd.did=D.did
```

result:

{(Smith, 401), (Smith, 403), (Smith, 401)}

Figure 3: Instances of the Relational and the Virtual Object-Oriented Database

Before describing the query translation algorithm, we first introduce the basic notions of 'equivalence of queries' and 'equivalence translations'.

3.1 Basic Definitions

As already indicated in the above example, in many cases there is no translation of an SOQL to an SQL query which provides exactly the same result. Therefore in this context we have to introduce a weaker notion of equivalence. Informally, our notion of *result equivalence* means that the SQL query produces an answer which may be easily converted into the desired result, particularly without further selection and join operations. The former ensures, that only the necessary amount of data will be transferred which is important for performance reasons, especially if the relational system is accessed via network, and the latter ensures, that the query can be answered by exactly one SQL statement.

Definition (Equivalence of queries)

Let RDB be the actual relational database with schema RS, ODB the virtual object-oriented database with schema OS, *res*(S, RDB) the resulting table when executing S on database RDB, and *res*(Q, ODB) the result expected from an execution of Q on ODB. Then we say, Q and S are *result equivalent* up to simple formatting operations if the following property holds:

$$f_Q(res(S, RDB)) = res(Q, ODB), (*)$$

where the formatting function f_Q is composed by structuring, grouping, projection, nesting and aggregate operations (c.f. section 4).

Based on the above definition, we are able to define the notion of an 'equivalence translation' from SOQL into SQL.

Definition (Equivalence translation)

Any mapping t, t: Q $\mapsto$ (S, f_Q) translating an SOQL query Q into a result equivalent SQL query S and providing a formatting function f_Q, such that (*) holds, is said to be an *equivalence translation*.

Note, that t is a partial mapping, because there are SOQL queries, that can not be translated into SQL. It would be desirable, however, for t to be *complete* in the following sense: If there exists an SQL query S' and a formatting function f'_Q with

$f'_Q(\, res(\text{S'}, \text{RDB})\,) = res(\text{Q}, \text{ODB})$

for a given query Q, then t should return a pair (S, f_Q) with the same property as S' and f'_Q.

Before presenting the translation algorithm t in detail, we will formalize the following helpful observation that allows a uniform treatment of chains of method applications:

Let $R_i := flat\_type(V.m_1.m_2. \ldots .m_i)$ be the flat class type resulting from the successive method application to V which may be uniquely determined since our schema transformation algorithm produces no subtype hierarchies. Chains $V.m_1.m_2. \ldots .m_n$ of method applications occurring within SOQL-statements may be divided into the first k and the last n-k+1 subchains, $0 \le k \le n+1$, such that: If $0 \le i < k$, then R_i is a non-basic class type (e.g. Passenger, Departure with a corresponding table in RS), and if $k \le i \le n$, then R_i is a basic class type (Boolean, String, Integer, ...). A short example will illustrate this fact:

p.departures.passengers.name where *p* ranges over class *Passenger* implies k=3, n=3 with
$R_0 = flat\_type(p) = Passenger$
$R_1 = flat\_type(p.departures) = Departure$
$R_2 = flat\_type(p.departures.passengers)$
$= Passenger$
$R_3 = flat\_type(p.departures.passengers.name)$
$= String$

This observation ensures that chains of method applications only have to be resolved until the first basic class type is encountered as implicitly used in transformation step 2 below. Loosely speaking, a chain $V.m_1.m_2. \ldots .m_{k-1}$ indicates a join sequence.

3.2 Translation Algorithm

By providing a step-by-step algorithm for the translation, in the following we constructively define an *equivalence translation t* which translates SOQL queries into *result equivalent* SQL queries. Since SOQL queries can be more structured than SQL queries, the result structure of an SOQL query needs to be flattened before it can be processed by the relational system. To build the desired result structure, a sequence of formatting operations is recorded during the flattening process (c.f. section 4). In the following, it is assumed that all class variables occurring in the 'for each' clauses of the query and all its subqueries have pairwise distinct names. Otherwise, they will be consistently renamed. New variables introduced during the transformation are denoted by V_i.

Before applying the steps of the translation algorithm, we transform the considered SOQL query into a nested set expression. The 'select' clause becomes the result part of the set. The range and class variable definitions of the 'for each' clause are transformed into 'element in set' relationships. The 'where' clause is syntactically adapted to the set notation and occurring subqueries are recursively transformed into corresponding set expressions. In figures 4 and 5, the translation process is illustrated using the query examples from section 2.

Step 1: Resolution of structured expressions and generalized dot-notation

In this step, structured expressions and chains of method applications using the generalized dot-notation are resolved. Chains of method applications

$$V.m_1. \ldots .m_n$$

in the 'select' or 'for each' clause are successively resolved as

$$(V.m_1). \ldots .m_n$$

if m_1 is a method defined for V and as

$$\{v.m_1 \mid v \in V\}.m_2. \ldots .m_n$$

if V is set-valued and m_1 is not defined for V. The translation of generalized dot-notation occurring in the 'where' clause is slightly different. In this case, an existential quantification is introduced (c.f. translation step 1 in figure 4). The translation of more complex conditions involving nested sets which can not be expressed in SQL are described in subsection 3.3. Note, that the translation is possible since chains of method applications have only to be resolved until the first basic class type is encountered (c.f. observation in section 3.1).

Structured expressions

$$V.\left[m_{1_1}. \ldots .m_{l_1}, \ldots, m_{n_1}. \ldots .m_{l_n}\right]$$

```
select  P.name, P.address
for each  Passenger P,  P.departures D
where  D.start = '06/18/93' and  D.airline.name = 'Lufthansa'
```

$\equiv$ {(P.name, P.address) | P $\in$ Passenger $\wedge$ D $\in$ P.departures $\wedge$ D.start = '06/18/93' $\wedge$ D.airline.name = 'Lufthansa' }

$\equiv_{(\text{step 1})}$ {(P.name, P.address) | P $\in$ Passenger $\wedge$ D $\in$ P.departures $\wedge$ D.start = '06/18/93' $\wedge$ $\exists$ V_1: V_1 = D.airline $\wedge$ V_1.name = 'Lufthansa' }

$\equiv_{(\text{step 3})}$ {(P.name, P.address) | $\exists$ V_1: P $\in$ Passenger $\wedge$ D $\in$ Departure $\wedge$ join(P, D) $\wedge$ D.start = '06/18/93' $\wedge$ V_1 $\in$ Airline $\wedge$ join(D, V_1) $\wedge$ V_1.name = 'Lufthansa' }

$\cong$

```
select  P.name, P.address
from  Passenger P, Departure D, Airline V1
where  join(P, D)  and  join(D, V1)  and  D.start = '06/18/93'  and  V1.name = 'Lufthansa'
```

$\cong$

```
select  P.name, P.address
from  Passenger P, Departure D, Airline V1, Pass_Dept V2
where  P.pid = V2.pid  and  V2.did = D.did  and  D.airline-id = V1.airline-id  and
       D.start = '06/18/93'  and  V1.name = 'Lufthansa'
```

Figure 4: Translation of Query Example 1

are also resolved successively as

$$\left(V.m_{1_1}.\ \dots\ .m_{l_1},\ \dots,\ V.m_{n_1}\dots .m_{l_n} \right)$$

if at least one of the m_{i_1} is directly applicable to *V* and as

$$\{ v.\left[m_{1_1}.\ \dots\ .m_{l_1},\ \dots,\ m_{n_1}.\ \dots\ .m_{l_n}\right] \mid v \in V\}$$

if *V* is set-valued and none of the m_{i_1} is defined on *V*. Note, that structured expressions and chains of method applications may be nested into each other. Therefore, both translation rules may have to be applied alternately.

Step 2: Resolution of complex range variables

In this step, variables ranging over arbitrary path expressions are replaced by variables ranging only over classes corresponding to relations. To select all passengers together with the sets of co-passengers for each of their flights, we may write

```
select  P.name, CP.name
for each   Passenger P,
    P.departures.passengers CP
```

In this case, the range variable CP ranges over sets of passengers which cannot be directly expressed in SQL. Therefore, the corresponding nested set expression

{(P.name, CP.name) | P $\in$ Passenger $\wedge$ CP $\in$ {V_1.passengers | $V_1$$\in$ P.departures}}

is translated into

{(P.name, V_1.passengers.name)| P $\in$ Passenger $\wedge$ V_1 $\in$ P.departures} $\equiv_{(\text{step 1})}$

{(P.name, {V_2.name | V_2 $\in$ V_1.passengers}) | P $\in$ Passengers $\wedge$ V_1 $\in$ P.departures}.

More formally, the translation can be expressed as

$\{(x, y)| x \in X \wedge y \in \{h(z) \mid z \in Z \wedge p(x, z)\} \wedge q(x, y)\}$

$\equiv\!>$ $\{(x, h(z)) \mid x \in X \wedge z \in Z \wedge p(x, z) \wedge q(x, h(z))\}$

with a subsequent resolution of generalized dot-notation (c.f. step 1).

Step 3: Resolution of object references

All remaining object references are resolved as follows.

V_1 *op* X.m $\equiv\!>$
V_1 $\in$ flat_type(X.m) $\wedge$ join(X, V_1),

where *op* = '$\in$' or '=' depending on whether X.m is set or single valued. In this step, join predicates join(X, V_i) are introduced with the intended meaning: join(X, V_i) is true if there is an object reference from X to V_i.

Note, that in the previous steps path expressions involving aggregate operations have not been resolved. In this step, however, we want to resolve possible object references that are part of such path expressions. Since the aggregate operations are applied to sets, we translate path expressions

$$V.m_1.\ \dots\ .m_n$$

with m_n being an aggregate operation into

$$\{v \mid v \in\ V.m_1.\ \dots\ .m_{n-1}\ \}.m_n.$$

Then all object references in $V.m_1.\ \dots\ .m_{n-1}$ can be resolved by join predicates as described above. In some cases, however, no additional joins may have to

select P.[name, address], P.departures.[[did, passengers.name], count]
for each Passenger P
where P.address like '%München% ($cond$:= 'P.address like '%München%')

$\equiv$ {([P.[name, address], P.departures.[[did, passengers.name], count]) | P $\in$ Passenger $\wedge$ $cond$}

$\equiv_{\text{(step 1a)}}$ {((P.name, P.address), ({(V_1.did, V_1.passengers.name) | $V_1 \in$ P.departures}, P.departures.count)) | P $\in$ Passenger $\wedge$ $cond$}

$\equiv_{\text{(step 1b)}}$ {((P.name, P.address), ({(V_1.did, {V_2.name | $V_2 \in V_1$.passengers}) | $V_1 \in$ P.departures}, P.departures.count)) | P $\in$ Passenger $\wedge$ $cond$}

$\equiv_{\text{(step 3)}}$ {((P.name, P.address), ({(V_1.did, {V_2.name | $V_2 \in$ Passenger $\wedge$ join(V_1, V_2)}) | $V_1 \in$ Departure $\wedge$ join(P, V_1)}, {V_1}.count)) | P $\in$ Passenger $\wedge$ $cond$}

$\cong_{\text{(step 4a)}}$ {((P.name, P.address), P.key, ((V_1.did, {V_2.name | $V_2 \in$ Passenger $\wedge$ join(V_1, V_2)}), {V_1}.count)) | P $\in$ Passenger $\wedge$ $V_1 \in$ Departure $\wedge$ join(P, V_1) $\wedge$ $cond$}

$\cong_{\text{(step 4b)}}$ {((P.name, P.address), P.key, ((V_1.did, V_1.key, V_2.name), V_1.key)) | P $\in$ Passenger $\wedge$ $V_1 \in$ Departure $\wedge$ join(P, V_1) $\wedge$ $V_2 \in$ Passenger $\wedge$ join(V_1, V_2) $\wedge$ $cond$}

$\cong_{\text{(step 4c)}}$ {(P.name, P.address, P.key, V_1.did, V_1.key, V_2.name, V_1.key) | P $\in$ Passenger $\wedge$ $V_1 \in$ Departure $\wedge$ $V_2 \in$ Passenger $\wedge$ join(P, V_1) $\wedge$ join(V_1, V_2) $\wedge$ $cond$}

$\cong$ select P.name, P.address, P.pid, V_1.did, V_2.name
from Passenger P, Departure V_1, Passenger V2, Pass_Dept V3, Pass_Dept V4
where P.pid = V3.pid and V3.did = V1.did and V1.did = V4.did and V4.pid = V2.pid
and P.address like '%München%'

Figure 5: Translation of Query Example 2

be introduced. In example 2, the *P.departures* comes from a structured expression that already has been resolved and, therefore, we do not need to repeat the part '$V_1 \in$ Departure $\wedge$ join(P, V_1)' but still use V_1.

The result of the three steps of the translation algorithm that have been described so far is semantically and structurally *equivalent* to the original query but with all dot generalizations and structured expressions being resolved. In the following steps, the result is changed either by adding attributes or by flattening the result structure. Still, our notion of *result equivalence* up to simple formatting operations is preserved since the necessary formatting operations are recorded.

Step 4: Resolution of nested result types

In this step, the nested structure of result tuples is resolved by shifting set conditions of the inner sets onto the outer level and adding key information. The translation is done level by level starting outermost-leftmost. Key information which is necessary to reconstruct the desired result structure is introduced for all variables on the outer level (c.f. step 4a and 4b in figure 5). At the same time, the formatting function which reconstructs the intended result structure successively (c.f. section 4) is extended by the inverse structuring, grouping, projection and nesting operations. Aggregate operations coming from inner nesting levels need to be removed (c.f. translation step 4b in figure 5). Formally, the flattening of one nesting level can be described as:

{(x, {y | y $\in$ Y $\wedge$ p(x, y)}) | x $\in$ X $\wedge$ q(x,)} $\Rightarrow$
{(x, key(x), y) | y $\in$ Y $\wedge$ p(x, y) $\wedge$ x $\in$ X $\wedge$ q(x)}.

This translation rule is applied until the nesting structure of the result tuple is flat. Then, only the remaining tuple structure needs to be flattened (c.f. translation step 4c in figure 5). Again, in this step the formatting function is extended by the inverse operations and key information is added to the result list instead of the omitted aggregate operations.

The remaining translation into a valid SQL query is straightforward provided we restrict SOQL conditions to permissible SQL conditions. More complex condition parts may also be translated into SQL. In subsection 3.3, some extensions of the condition part are described that can be translated into permissible SQL statements. Note, that replacing the join predicates join(R, S) may introduce additional relations which are necessary, e.g. Pass_Dept in example 2, to establish m:n relationships.

3.3 Extensions of the Condition Part

Since SOQL has more expressive power than SQL, there are some cases where SOQL queries do not have *result equivalent* SQL queries. However, as we will show in the following, the condition part that is permissible in SOQL queries while still guaranteeing an equivalence translation can be extended considerably. Simpler extensions, for example, are methods on set types such as 'el in set' which may be replaced by computing the set in a subquery and applying the corresponding SQL constructs 'el in (select ...)' to the result of the subquery. Some important extensions to be included into the translation algorithm are:

Extension 1: Generalized dot-notation in conditions

In the condition part of SOQL queries, set-valued method path expressions like D.passengers.name ='Jones' may occur at all positions where the SQL syntax only allows simple column expression like P.name='Smith'. According to the definition of the semantics of method path expressions, the resolution of set-valued method path expressions in conditions would result in a set of booleans which has to be 'flattened' to a single boolean value.

$\{x \mid x \in X\}$ *op* y is defined by $\exists x: x \in X \wedge x$ *op* y, if '*op* y' is not applicable to the whole set.

Example:

D.passengers.name = 'Jones' $\equiv>$
$\{V_1$.name $\mid V_1 \in$ D.passengers$\}$ = 'Jones' $\equiv>$
$\exists V_1$: $V_1 \in$ D.passengers $\wedge$ V_1.name = 'Jones'

Only applying the resolution according to the generalization of dot-notation to

$\{V_1$.name $\mid V_1 \in$ D.passengers$\}$ = 'Jones'

results in

$\{V_1$.name = 'Jones' $\mid V_1 \in$ D.passengers$\}$

which is a set of booleans. Like in IRIS [Fis 89], in SOQL sets of booleans in conditions are implicitly 'or'-connected [KKM 93b] evaluating to true if at least one element is true.

Extension 2: Set inclusion

Inclusion conditions $A \subseteq B$ with $A = \{x_1 \mid p(x_1)\}$ and $B = \{x_2 \mid q(x_2)\}$ in the SOQL 'where' clause may be transformed in the following way, provided *A* and *B* can be processed by SQL subqueries.

$A \subseteq B \equiv>$ not exists$\{x_1 \mid p(x_1)$ and
not exists $\{x_2 \mid q(x_2) \wedge x_2 = x_1\}$ $\}$

Example:

D1.passengers.name $\subseteq$ D2.passengers.name
$\equiv>$ not exists{P1.name | P1 $\in$ Passenger $\wedge$
join(P1, D1) $\wedge$ not exists
{P2.name | P2 $\in$ Passenger $\wedge$
join(P2, D2) $\wedge$ P2.name=P1.name} }

Extension 3: Union, intersection, difference

Predicates like $x \in A \cup B$, $x \in A \cap B$, $x \in A - B$ in the SOQL 'where' clause can be transformed to

x in $\{x_1 \mid p(x_1)\}$ or x in $\{x_2 \mid q(x_2)\}$,
x in $\{x_1 \mid p(x_1)\}$ and x in $\{x_2 \mid q(x_2)\}$,
x in $\{x_1 \mid p(x_1)\}$ and x not in $\{x_2 \mid q(x_2)\}$,

which can be transformed to SQL, if A and B can be transformed to valid SQL subqueries.

Note, that the subqueries A and B in extensions 2 and 3 may only return unstructured results since otherwise the nesting operators of SQL are not applicable. Serious problems in the query translation process may be caused by user extensions to the object-oriented schema, such as additional attributes or user-defined methods. In the case of using user-defined methods in an SOQL query, the data necessary to evaluate the query has to be retrieved iteratively from the relational system before it can be used to execute the methods. If classes are extended by additional attributes, the data necessary to evaluate the condition part of a query is retrieved partially from the relational system and the additional data is retrieved from the system managing the additional data. According to the extended object-oriented schema, the corresponding data of both sources is related to each other before the condition is evaluated and the desired data is retrieved as specified in the 'select' clause. In both cases, it may be necessary to transfer large amounts of data, even in cases where the resulting data set is rather small and, therefore, performance problems may occur. Note, that the problems are only caused in cases where there is no corresponding SQL query.

4. Transformation of the Result

As already mentioned, to automatically restructure the result flattened by the last steps of the translation algorithm a formatting function has to be generated. In each partial transformation of these steps, formatting primitives are recorded which are composed in the reverse order of their creation, such that the last primitive is applied first to the result returned by the generated SQL query.

This means for query example 2 in figure 5, that the result returned by the final SQL query has to be structured into subtuples after copying the V_1.key attribute twice to revert the last two steps. Then, the tuples are partitioned into groups by equal values of the attribute combination P.key, V_1.key and for each group, the values of V_2.name are combined to form the inner sets, i.e. for each person and one of its departures, all passengers belonging to this departure

$f_0 = \varepsilon$

f_1 = *project(*P.name, P.address, V_1.did, V_2.name, (V_1.key*)*.count*)* *(step 4a)*

f_2 = *group(by*(P.key), *nest*(V_1.did, {V_2.name}), *count*(V_1.key*))* *(step 4a,4b)*

f_3 = *project(*P.name, P.address, P.key, V_1.did, {V_2.name}, V_1.key*)* *(step 4b)*

f_4 = *group(by*(P.key, V_1.key*), nest*(V_2.name)*)* *(step 4b)*

f_5 = *structure((*P.name, P.address), P.key, ((V_1.did, V_1.key, V_2.name), V_1.key*))* *(step 4c)*

f_6 = *structure*(P.name, P.address, P.key, (V_1.did, V_1.key, V_2.name), V_1.key) (step *4c*)

f_7 = *project(*P.name, P.address, P.key=P.pid, V_1.did, V_1.key=V_1.did, V_2.name, V_1.key=V_1.did*)* *(final step)*

Figure 6: Formatting Function for Query Example 2

are grouped into a set. Now, the first V_1.key can be projected out. Next, the intermediate result is grouped by P.key to be able to count each person's departures and to combine the information on each person's departures into a set. After projecting out the second V_1.key and P.key, the correct answer in the desired result structure is reached.

In the following, the formatting primitives are defined as generic functions which may be arbitrarily composed:

- *structure (attr_list$_1$, ..., attr_list$_l$)*: *Res* $\rightarrow$ *Res*
  combines the attributes of each *attr_list$_i$* into a tuple $(a_{i1} ..., a_{in_i})$ and the whole expression itself into a tuple $((a_{11}, ..., a_{1n_1}), ..., (a_{l1}, ..., a_{ln_l}))$.

- *group (by(attr_list$_1$), op(attr_list$_2$), ..., op(attr_list$_l$))*: *Res* $\rightarrow$ *Res*
  groups *Res* according to equal values of *by(attr_list)*. For each group, the attributes listed in *op(attr_list)* are combined to sets if *op* = nest, and aggregated using the corresponding aggregate operator if *op* = min, max, count, avg, sum. In order to get only one value per group, all attributes that occur in none of the *attr_list* have to be functionally dependent on the attributes in *by(attr_list)*.

- *project(a_new$_1$=a_old$_1$, ..., a_new$_l$=a_old$_l$)*: *Res* $\rightarrow$ *Res*
  projects *Res* onto the specified attributes allowing attributes to be duplicated and renamed. The assignment *a_new$_i$* = *a_old$_i$* is only needed if attributes are duplicated or renamed.

The formatting function is constructed as concatenation of the formatting primitives:

$$f_Q = f_0 \circ f_1 \circ \cdots \circ f_{(n-1)} \circ f_n$$

with f_0 being the ε–function. The formatting primitives and their concatenation to the formatting function f_Q are illustrated in figure 6 using query example 2.

5. Summary and Conclusions

A major challenge in building a CIS is the integration of existing ISs. We propose the concept of a cooperative interface to support the cooperation of isolated systems without changes to these systems. The main contribution of this paper is the query translation algorithm which is an important part of such an interface. Our algorithm allows an automatic translation of SOQL queries issued against the created object-oriented schema into 'result equivalent' SQL queries for the original relational schema. In the query translation algorithm, first chains of method applications are replaced by appropriate joins and subqueries on the relational side, the conditions are replaced by equivalent SQL conditions and, since SQL cannot provide structured results, the nested structure of the result is flattened but enhanced with additional key information. Simultaneously, the inverse formatting operations are created allowing reconstruction of the desired result from the result of the SQL query.

We believe that our query translation algorithm is easily applicable and thus, of high practical importance. It does not require any change to the relational system, the data or existing applications and therefore, our schema transformation and query translation algorithm is a practical solution for the development of cooperative systems. The implementation of the schema transformation and operation translation algorithms with complete support of user-defined methods and additional object-oriented classes is currently on the way, but not yet finished. One open problem is the optimization of queries which involve user extensions to the schema, complex set operations or arbitrarily structured results. In such SOQL queries which have no one-to-one correspondence to an SQL query, the query optimization cannot be done on the relational side. Therefore, we have to optimize the query execution plan to reduce the amount of data which needs to be transferred between the object-oriented query interface and the relational system. Performance issues will be of high importance for such a system to be used in real world applications.

In our future work, we plan to extend the schema enrichment and query translation algorithms to cover the automatic detection and creation of subtype hierarchies and to deal with complex methods. We will try to find possibilities to translate complex conditions involving set operations on structured results. We will further work on the optimization issue trying to provide an acceptable performance even in complicated cases. Finally, we will investigate on the distribution of global queries and the combination of their results.

References

[ASL 89] Alashqur A. M., Su S. Y., Lam H.: *'OQL: A Query Language for Manipulating Object-oriented Databases',* Proc. 5th Int. Conf. on Very Large Data Bases, Amsterdam, 1989, pp. 433-442.

[BC 92] Brodie M. L. Ceri S.: *'Intelligent and Cooperative Information Systems',* in: [EJ 92], 1992.

[BCD 92] Bancilhon F., Cluet S., Delobel C.: *'A Query Language for O_2'*, chapter 11 in: [BDK 92], 1992, pp. 234-255.

[BDK 92] Bancilhon F., Delobel C, Kanellakis P. (eds.): *'Building an Object-Oriented Database System - The Story of O_2'*, Morgan Kaufmann, San Mateo, CA, 1992.

[BHP 92] Bright M. W., Hurson A. R., Pakzad S. H.: *'A Taxonomy and Current Issues in Multidatabase Systems',* Proc. IEEE Computer, 1992, pp. 50-60.

[EJ 92] Ellis C. A., Jarke M.: *'Distributed Cooperation in Integrated Information Systems',* Proc. 3rd Int. Workshop on Intelligent and Cooperative Information Systems, Dagstuhl, Germany, in: Aachener Informatik-Berichte, 92-18, 1992.

[Fis 89] Fishman D. H. et al: *'Overview of the Iris DBMS'*, chapter 10 in: *Object-Oriented Concepts, Databases and Applications* by Kim W. and Lochovsky F.H. (eds.), ACM Press Frontier Series, Addison Wesley, Reading, MA, 1989, pp. 219-250.

[HD 91] Harris C., Duhl J.: *'Object SQL'*, chapter 11 in: *Object-Oriented Databases with Applications to CASE, Networks, and VLSI Design* by Gupta H. and Horowitz E., Prentice Hall, 1991, pp. 199-215.

[Heu 89] Heuer A.: *'Equivalent Schemes in Semantic, Nested Relational, and Relational Database Models',* Proc. 2nd Symp. on Mathematical Fundamentals of Database Systems, Visegrád, Hungary, 1989, in: Lecture Notes in Computer Science, Vol. 364, Springer, 1989, pp. 237-353.

[Inf 92] Informix: *'INFORMIX - TP/XA'*, Informix Software Inc., Menlo Park, CA, 1992.

[Ing 92] Ingres: *'INGRES / Star',* Ingres, Frankfurt, 1992.

[ISO 92] ISO/IEC: *'Database Language SQL'*, ISO/IEC 9075:1992 (German Standardization: DIN 66315).

[KKM 93a] Keim D. A., Kriegel H.-P., Miethsam A.: *'Integration of Relational Databases in a Multidatabase System based on Schema Enrichment',* Proc. Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS), Vienna, Austria, 1993, pp. 96-104.

[KKM 93b] Keim D. A., Kriegel H.-P., Miethsam A.: *'Object-Oriented Querying of Existing Relational Databases',* Proc. 4th. Int. Conf. on Database and Expert Systems Applications (DEXA'93), Prague, Czech Republic, 1993, in: Lecture Notes in Computer Science, Vol. 720, Springer, 1993, pp. 325-336.

[KL 89] Kim W., Lochovsky F.H.: *'Object-Oriented Concepts, Databases and Applications'*, ACM Press Frontier Series, Addison Wesley, Reading, MA, 1989.

[Ora 92] Oracle: *'ORACLE SQL*Connect',* Oracle, München, 1992.

[RPR 89] Reddy M. P., Prasad B. E., Reddy P. G.: *'Query Processing in Heterogeneous Distributed Database Management Systems',* in: Integration of Information Systems: Bridging Heterogeneous Databases, Amar Gupta (ed.), 1989, pp. 264-277.

[SL 90] Sheth A. P., Larson J. A.: *'Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases',* ACM Computing Surveys, Vol. 22, No. 3, 1990, pp. 183-236.

[Syb 90] SYBASE: *'Connectivity: Technical Overview',* Sybase Inc., Emeryville, CA, 1990.