Object-oriented views of relational databases incorporating behaviour

Mark W.W. Vermeer
University of Twente
vermeer@cs.utwente.nl

Peter M.G. Apers
University of Twente
apers@cs.utwente.nl

Abstract

The derivation of object-oriented (OO) views of relational databases has been a important research topic, in particular in the context of federated databases using an OO-model as the common datamodel. So far, such views have concerned data structures only. We argue that it is meaningful to extend the scope of OO-views to include *methods* and *constraints* derived from applications on the underlying schemata. The views can thus be equipped with methods representing common queries on the underlying schemata and application-enforced constraints, enhancing the semantics of the schema, yielding a higher-level interface to applications, and supporting the possibility of a gradual migration to an OODB. We focus on the derivation of methods from queries here.

1 Introduction

The derivation of views of relational databases expressed in terms of a semantically stronger data model has traditionally been investigated in the context of database re-engineering. Recently, there has been a renewed interest in this subject due to its applicability in federated databases [10], where a common data model is used to cope with heterogeneous component databases. Object-oriented models are deemed to be of particular use in this context, due to their semantic expressiveness and the possibility to hide implementation details using encapsulation.

Moreover, while the OO-paradigm has proven its relevance to complex application domains, at present the most mature DBMS-technology is still based on the relational datamodel. Constructing an object-oriented view of a relational database can then be seen as a first step of a *migration* from relational to OO data management. Subsequent applications are developed for the object schema, while the actual migration of data from an RDBMS to an OODBMS is postponed to a later stage.

Up to the present, research on the (semi-)automatic construction of such views has been focussing on deriving object *structures* translated from the schema of the underlying relational database. The incorporation of behaviour, in terms of *operations* and *constraints* has not yet been considered. In the relational context, behaviour of data items must be determined from application software acting upon the schema, whereas an OO-model explicitly incorporates both structure and behaviour. In our opinion, equipping OO-views with methods and constraints derived from application software is meaningful for the following reasons:

(1) Specifying operations and constraints on the data structures enhances the semantics of the schema. Especially in non-administrative application domains, behaviour of objects is an important aspect of schema semantics.

(2) Offering a high-level interface to the underlying data consisting of operations defined in existing applications is essential to the efficient development of new applications acting on the view. Deriving object methods from legacy applications supports code reuse, one of the traditional advantages of the OO-paradigm.

(3) In case data is migrated gradually from the legacy database to an OODB, the interface offered by the operations defined on the view can provide the necessary location transparency, allowing data to be manipulated without concerning the phase of migration of individual objects.

The TM language

We define OO-views using the TM [1,6] object-oriented database specification language, because of its high level of abstraction and its expressiveness. TM allows the specification of data structures in terms of Classes and Sorts (the latter being abstract data types without object identity), and provides a computationally

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995
© World Scientific Publishing Co. Pte Ltd

complete, functional data manipulation language for method and constraint specification, which is also capable of expressing set-oriented queries. A large and well-defined subset of TM-specifications is executable; the mapping to an OO-PL is performed at the implementation stage.

Scope of the paper

The focus of this paper is the translation of SQL-queries to TM-methods, given a translation of the relational schema the query is defined on, into a corresponding TM-schema. Although we use TM as our target language, our translation algorithm could easily be adapted for any set-oriented OO query language. To arrive at a fully equipped OO-view, however, it is necessary to address translating general application code as well, deriving constraints from applications. These issues will be the subject of subsequent research.

Since the DML of TM has more expressive power than SQL, we could provide a direct translation for any SQL-query, simulating relational concepts on the object structure by using joins to establish relationships etc. Obviously, the direct translation to a semantically stronger language is not interesting in general. We therefore focus on the gain of semantics that can be achieved by exploiting additional information available from the OO-schema.

Related work

To our knowledge, the only publication specifically addressing the translation of SQL-queries to an object-oriented DML is the work of Meng et al. [9]. It is devoted to the run-time translation of SQL-queries on a relational front-end to an object-oriented database system. Although there are some strong similarities between their translation algorithm and ours, which we developed independently, two important differences can be pointed out.

Firstly, in the context of [9], there is a very simple mapping from the relational to the object-oriented schema, as the former is a translation of the latter rather than the other way around. In our context, due to the extra semantic information that must be added during the translation from a relational to an object-oriented schema, there is a non-trivial mapping from tables to classes. We show that the translation of SQL-queries benefits considerably from these additional semantics.

Secondly, since [9] focusses on the operational aspects of query translation, they do not discuss the translation of nested queries, as there exist unnesting algorithms [8] that transform nested queries into an equivalent collection of simple queries. This approach would be inadequate for our purposes, as it would obscure the meaning of the original query.

Note that existing work on query translation in FDBSs such as [4,9,11], is devoted to the run-time translation of queries on the view in terms of queries on the underlying databases. Our translation works the other way round and focusses on semantics rather than efficiency.

Existing work on translating data structures is still relevant in our approach. We presuppose a sufficiently powerful schema translation, using methodologies such as those of Saltor and Castellanos [2], Johannesson [7], or Chiang et al. [3]. Although these methods themselves are somewhat different, their output schemata for a given relational schema are roughly equivalent.

Overview

The remainder of this paper is organized as follows. Section 2 describes the schema mapping information needed in the query translation process. In Section 3 we describe two translations of an SQL-query, illustrating the idea of semantic expressiveness of TM-methods. Next, in Section 4 we describe how SQL-queries and object schemata can be matched, resulting in the definition of the object structure described by the query, which can subsequently be used to determine the class or classes to which the translated methods should be assigned. Finally, Section 5 describes the translation of SQL-queries to TM-methods. Section 6 presents a brief summary of the translation algorithm.

2 Schema translation

Our query translation method assumes that the relational schema on which the input query is defined has been translated into an object schema. Several methods for the (semi-)automatic translation of a relational schema to a semantically richer model have been described. We discuss characteristics of such methods and requirements we place on their output schema using an example relational schema.

2.1 Existing methodologies

We assume a semantically powerful translation procedure [2,3,7] is used to translate the relational schema into an object structure. Such methods are capable of inferring a *class hierarchy*, and detect so-called *missing entities* which were implicit in the original schema. Required input information usually consists of table definitions in 3NF, information on keys, and inclusion

dependencies between attribute values of different tables. Figure 1 shows an example input to a schema translation algorithm.

```
EMPLOYEE(ssn, name, address, sex, sal, superssn, dno)
DEPARTMENT(dnumber, dname, mgrssn, mgrstdate)
DEPENDENT(essn, dependent-name, sex, bdate, relship)
PROJECT(pnumber, pname, plocation, dnum)
WORKS-ON(wssn, pno)

EMPLOYEE.superssn ⊆ EMPLOYEE.ssn
EMPLOYEE.dno ⊆ DEPARTMENT.dnumber
DEPARTMENT.mgrssn ⊆ EMPLOYEE.ssn
DEPENDENT.essn ⊆ EMPLOYEE.ssn
PROJECT.dnum ⊆ DEPARTMENT.dnumber
WORKS-ON.wssn ⊆ EMPLOYEE.ssn
WORKS-ON.pno ⊆ PROJECT.pnumber
```

Figure 1: Example relational schema with inclusion dependencies

Application of any of these methods results in an object schema, where each of the classes is a mapping of a number of attributes from one or many tables. Figure 2 shows a possible translated schema, expressed in TM (the ℙ-symbol represents a set-valued attribute).

Note the non-trivial mapping from tables to classes. Whereas EMPLOYEE is mapped to a single class, DEPARTMENT has been split into the classes Department and Manager, the latter being a *missing entity*, detected due to the extraneous secondary key mgrssn. The WORKS-ON relationship table does not have a corresponding class, but has been translated into the set-valued object references Employee.projects and Project.employees. DEPENDENT has been translated as a Sort instead of a Class, since it represents a *weak entity* depending on EMPLOYEE for its existence. Furthermore, an **isa**-relationship between Manager and Employee has been detected, based on the inclusion dependency defined on their key attributes.

The schema translation thus results in a definition of an object schema where each class C_k can be seen as a representation of a natural join on the common key attributes of a number of table fragments $\Pi_{a_{i1}\ldots a_{in}}(T_i)$. We say that C_k is an *image* of each of the tables T_i.

2.2 Documenting the schema mapping

As a formal representation of the mapping induced by the schema translation, we annotate the class definition of C_k with the predicates Image(className, tableName, attributeSet) reflecting the set of table attributes the class represents, Key(className, attributeSet), indicating which attributes form a key for the class, and DenotedBy(objectReferencingAttribute, foreignKeySet), indicating which foreign key or relationship table an object reference is based on.

```
Class Employee
attributes  ssn            : string
            name           : string
            address        : string
            sex            : string
            salary         : real
            employee       : Employee
            department     : Department
            dependents     : ℙDependent
            projects       : ℙProject
object constraints
oc1: forall x in projects | self in x.employees
end Employee

Class Department
attributes  dnumber        : integer
            dname          : string
            manager        : Manager
object constraints
oc1: manager.department = self
end Department

Class Manager isa Employee
attributes  mgrstdate      : Date
            department     : Department
object constraints
oc1: department.manager = self
end Manager

Class Project
attributes  pnumber        : integer
            pname          : string
            plocation      : string
            mdepartment    : Department
            employees      : ℙEmployee
object constraints
oc1: forall x in employees | self in x.projects
end Project

Sort Dependent
type   ⟨    dependent-name string
            sex            : string
            relationship   : string
            bdate          : Date ⟩
end Dependent
```

Figure 2: Translation of the example relational schema

```
Image(Employee,EMPLOYEE,{*})
Key(Employee,{ssn})
DenotedBy(employee, {superssn})
DenotedBy(department, {dno})
DenotedBy(projects, {WORKS-ON.pno})
```

Figure 3: Some annotations for the example TM schema

Figure 3 shows some of the annotations necessary for the example.

The input to the query translation algorithm thus consists of an object schema, obtained by applying a schema translation method to a relational schema, and some annotations on the schema mapping induced by the translation. The next section discusses the desired output of such an algorithm.

3 Semantic expressiveness of query translations

Assuming a translated schema, annotated as above, we now turn to the translation of SQL-queries on the original relational schema to methods on the object schema. These methods are expressed in TM. Its most important statement in this context, which we will use for translating the SQL **SELECT**-statement, is the following:

collect e_1 **for** *var* **in** e_2 [**iff** *Pred*]

where e_2 is a set or list expression, *var* is a variable ranging over the elements of e_2, e_1 is a result expression of arbitrary type (typically involving *var*), and *Pred* represents an optional condition under which e_1 should be retrieved.

Such a DML allows the translation of any SQL statement in an *ad-hoc* way; i.e. without exploiting any of the additional semantics captured by the object schema. In contrast, our translation method uses this additional semantics to achieve a more meaningful query translation. In particular, we exploit the class hierarchy and object references of the object model to eliminate joins, sometimes leading to the elimination of complete subqueries.

Example 3.1 Consider the query of Figure 4. Note that the intended meaning of the query, "List the names of managers not having a female dependent", is far from obvious in the relational schema. Note also how the definition of an employee which is also a manager, the latter being an entity that remains implicit in the relational schema, is expressed by the second subquery.

Figure 5 shows two example translations of the query. Note that where the first translation is relatively close to the SQL-form, the second is more in accordance with the semantics of the query and the object model. The subquery identifying employees that are also managers is made irrelevant by simply defining the method on members of class `Manager`. It is this kind of translation that is accomplished by our algorithm. □

```
SELECT  name
FROM    EMPLOYEE
WHERE   NOT EXISTS  (SELECT  *
                     FROM    DEPENDENT
                     WHERE   ssn=essn
                     AND     sex="F")
        AND
        EXISTS      (SELECT  *
                     FROM    DEPARTMENT
                     WHERE   ssn=mgrssn )
```

Figure 4: Example query

```
database retrieval method
Trans1(out ℙ⟨ Name : string ⟩)=
  collect
  ⟨ Name = x.name ⟩
  for x in employees
  iff  not exists y in x.dependents | y.sex="F"
  and exists z in departments |
              z.manager.mgrssn = x.essn

class retrieval method for Manager
Trans2(out ℙ⟨ Name : string ⟩)=
  collect
  ⟨ Name = x.name ⟩
  for x in self
  iff not exists y in x.dependents | y.sex="F"
```

Figure 5: Two translations of the example query

Sketch of the translation algorithm

As shown in the example, a good query translation should be based on a comparison of the semantics of the original query to the object model. In the translation of the SQL-query we can then use object references and the class hierarchy, which both can be seen as the *materialization* of certain joins in the relational schema.

We therefore *match* the join structure of an SQL-query Q to the structure of the object schema, thus obtaining a set of *complex object definitions* implied by Q. Selections on these complex objects are implemented by methods on their *root class*. The result of Q is then defined as a selection on the cartesian product of these complex object definitions.

The translation is complicated by the fact that both object references and nesting structures in SQL have *directions*, and the non-trivial mapping that may exist between tables and objects, as demonstrated in the example schema translation. We first discuss the matching process resulting in the identification of the complex object addressed by an SQL-query.

4 Identifying complex object definitions in a query

In this section, we describe how the semantic structure of object schemata and SQL-queries can be formalized and subsequently be matched, resulting in the set of complex object definitions described by the query. We do not elaborate on proving the correctness of our approach; the interested reader is referred to [12].

4.1 Representing object schemata and SQL-queries

For an object schema resulting from applying a translation algorithm to an underlying relational schema, we describe a representation formalism called the join materialization graph (JMG). A JMG describes the structure of the object schema, consisting of the class hierarchy and the delegation structure (the set of object referencing attributes defined in a schema). It further relates each structural relationship in the object schema to a pair of attributes from the underlying relational schema, on which a join must be performed to calculate the corresponding relationship for the relational schema.

The idea is that an object reference r from a class C_1 imaging a table T_1 to a class C_2 imaging a table T_2 with key K, where r is based on the foreign key F occurring in T_1, materializes the join $T_1 \bowtie_{F=K} T_2$. Furthermore, pairs of set-valued object references materialize joins involving a relationship table, and **isa**-relationships materialize key-based joins.

A *Join Materialization Graph (JMG)* consists of a set of vertices V, where each vertex represents a class or node from the object schema, and a set of labeled, directed edges $E = E_{fk} \cup E_{tb} \cup E_{me} \cup E_{isa}$, representing object references based on foreign keys (fk), relationship tables (tb), multiple entities with a common original table (me), and a super/subclassing relation (isa). Edges are labelled with the attribute pairs involved in the condition of the join they represent.

Example 4.1 Figure 6 shows the JMG for the example object schema. Notice the isa-edge from `Manager` to `Employee`, and the me-edge between `Manager` and `Department` due to their common origin: the

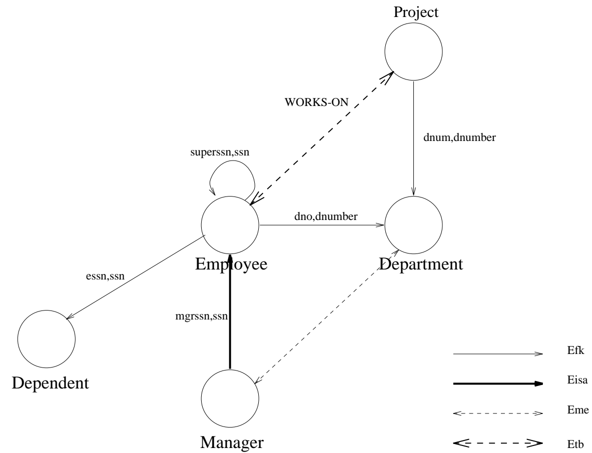


Figure 6: The join materialization graph for the example

`DEPARTMENT`-table. Notice also that we depict E_{tb} and E_{me}-edges as bidirectional, as they inherently occur in pairs, and we thus may traverse them in any direction. □

Following Dayal in [5], we use a comparable representation formalism for SQL-queries, including queries involving the nesting predicates **IN, NOT IN, EXISTS**, and **NOT EXISTS**. Dayal shows that **IN** and **EXISTS** conditions can be represented by semijoins, whereas **NOT IN** and **NOT EXISTS** conditions correspond to a special kind of antijoins. Such antijoins should always be computed after the computation of the regular joins. In the remainder of this paper, we shall restrict our discussion of antijoins to these cases. We assume conjunctive queries; the extension to disjunctive queries is analogous to [5].

The *Query Join Graph (QJG)* for a query Q consist of a set V of vertices representing tuple variables, and a set of labelled edges $E = E_{jn} \cup E_{sj} \cup E_{aj}$, where E_{jn}-edges are undirected and represent regular joins specified by Q, directed E_{sj}-edges represent semijoins and directed E_{aj}-edges represent the special antijoins described above. Labels contain attribute pairs appearing in join conditions.

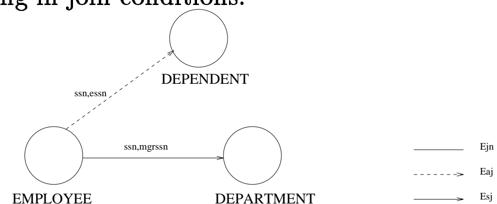


Figure 7: The query join graph for the example query

Example 4.2 Figure 7 shows the QJG associated with the example query from Figure 4. The example QJG is connected, and the subqueries are correlated. □

4.2 Matching SQL-queries to object schemata

This subsection discusses how the above representations can be matched to obtain a set of complex object definitions by Q, which will form the basis for our query translation procedure.

4.2.1 Tuple variables and the JMG

To perform such a matching, the focus of both representations must be brought into accordance. Note that where SQL-queries describe relevant tuples, i.e. *instances* of concepts described by a table, an object schema describes *classes*, i.e. the concepts themselves. As a consequence, Q may contain multiple tuple variables referring to the same table. To match the QJG to the JMG, we therefore consider JMG-nodes to represent *class instance variables* rather than classes themselves, where multiple class instance variables may be associated with a particular class. This motivates the expansion of a JMG given a query Q to a JMG_Q, where each class imaging a table addressed by multiple tuple variables in Q is replicated accordingly.

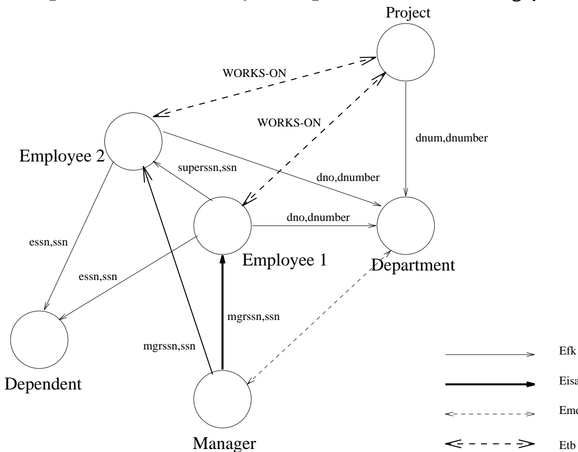


Figure 8: An example JMG-expansion

Example 4.3 Consider the following query on the relational schema of Figure 1.

SELECT X.name, Y.name
FROM EMPLOYEE X, EMPLOYEE Y, DEPARTMENT
WHERE X.ssn=Y.superssn **AND** X.dno=dnumber
AND dname="R&D"

Figure 8 shows the JMG_Q obtained from the JMG of Figure 6 for this query. □

Note that if Q does not contain multiple tuple variables referencing the same table, then $JMG_Q = JMG$. To keep formulations simple, in the remainder of this section we will assume that each tuple variable of Q addresses a unique table, and thus $JMG_Q = JMG$. We thus speak of tables instead of tuple variables and classes instead of class instance variables. The ideas discussed extend trivially to the general case.

4.2.2 Using Complex Object Definitions

A complex object definition is defined as follows.

Definition 4.1 : Complex Object Definition (COD) A *complex object definition* associated with a query Q on a relational schema R that has been translated to an object schema O, is a connected subgraph $\langle V', E' \rangle$ of the $JMG_Q\langle V, E\rangle$ induced by Q and O such that:

(1) each node $v \in V'$ represents a class C that images a table T appearing in the **FROM**-clause of Q.

(2) each edge $e \in E'$ is either an E_{me}-edge or represents the materialization of a join specified by Q.

(3) there exists a node $v_R \in V'$ representing a class C_R called the *root class*, from which all other $v' \in V'$ are reachable.

(4) for each object o in a class C represented by a node $v \in V'$ that is the image of a tuple t in the table T imaged by C, the following holds: If t is selected by Q, then there exists an object o_R in C_R from which o is reachable by following object references. □

This definition ensures that for each complex object definition, we can translate the part of Q addressing the COD into a method on its root class, selecting at least all relevant objects in the COD. The idea of the translation process is then to define a set S of CODs of minimal cardinality, such that Q can be translated to a database method M which combines the results of a set of methods S_M on the root classes of S. The CODs in S must therefore contain all classes relevant to Q.

Note that any class imaging a table addressed by Q is a trivial example of a COD satisfying this definition. An initial choice for S therefore consists of the set of classes addressed by Q. We then reduce the cardinality of S by connecting its elements using object references that match joins, semijoins, or antijoins of Q, resulting in a new, larger COD.

4.2.3 Matching joins, semijoins, and antijoins

Matching a regular join

We first describe matching a regular join appearing in Q. The simple idea is to match each edge e = (R,S, ⟨ R.A,S.B⟩) $\in QJG$ by an edge $e' = (C_1, C_2, \langle A, B\rangle) \in JMG_Q$ (disregarding directions, as e is undirected in this case), where C_1 and C_2 are images of R and

S, respectively. However, the match need not be so straightforward.

A join R$\bowtie_{A=B}$S can be matched by a chain of object references $r_1, r_2, ..., r_n$ where r_1 is a materialization of R$\bowtie_{A=A_1}$ R_1, r_2 materializes R_1 $\bowtie_{A_1=A_2}$ R_2,..., and r_n materializes R_{n-1} $\bowtie_{A_{n-1}=B}$S. Without loss of generality, we assume that in such cases there exists a (derived) object reference r that materializes R$\bowtie_{A=B}$S directly. On the other hand, the joins R$\bowtie_{A=B}$S$\bowtie_{B=C}$T may be matched by an object reference r materializing the join R$\bowtie_{A=C}$T (which is implied by transitivity) and s materializing either R$\bowtie_{A=B}$S or S$\bowtie_{B=C}$T. We therefore compute the transitive closure of the QJG, matching each of the joins implied by a query with a materialized join in the object schema. Finally, E_{tb} edges match any join condition involving an attribute of the connecting table they represent, since any join involving a table that implements an m:n relationship implies traversing this relationship.

Matching a semijoin

The *semijoin* R$\ltimes$S is defined as Π_R(R$\bowtie$S). A semijoin can therefore be matched like a normal join, regardless of the direction of the materialization; the projection does not play a role here. Also, in the eventual TM-translation there is no need for nesting. We illustrate this with a small example.

Example 4.4 Consider a query Q =

SELECT * FROM R **WHERE** R.A **IN**
  SELECT S.B **FROM** S.

Q has a QJG consisting of nodes for R and S and one edge representing R$\ltimes$S. Let C_1 and C_2 be images of R and S, respectively. Let r be a C_1-valued object reference in C_2 materializing the join S$\bowtie_{A=B}$R. Although the direction of r is opposite to that of the semijoin, the latter can simply be translated by collecting all C_1-objects referenced by C_2 objects. Q therefore describes a complex object definition with root C_2, on which the following retrieval method is defined:

collect x.r **for** x **in self**

□

Matching an antijoin

The *antijoin* R$\triangleright$S is defined as R$\setminus\Pi_R$(R$\bowtie$S). Here the direction of the materialization does play a role. Again we illustrate this with an example.

Example 4.5 Consider a query Q' =

SELECT * FROM R **WHERE** R.A **NOT IN**
  SELECT S.B **FROM** S **WHERE** S.C=0

Q' has a QJG consisting of nodes for R and S and one edge from R to S representing R$\triangleright$S. As in Example 4.4, assume that r is a C_1-valued object reference of C_2, materializing the join S$\bowtie_{A=B}$R. Here we cannot regard Q' as describing a complex object definition with root C_2. The semantics of this query construct is such that we cannot access every C_1-object satisfying Q from C_2, violating the fourth condition of Definition 4.1. The problem is that any C_1-object *not* referenced by a C_2-object satisfies Q.

Q' thus describes two CODs (each consisting of a single class) and is translated to the database method

collect x **for** x **in** C_1 **iff** x **not in**
  collect y.r **for** y **in** C_2 **iff** y.c=0

On the other hand, if r is a C_2-valued object reference of C_1, Q' can be translated as the following retrieval method on C_1 (assuming r is a single-valued attribute):

collect x **for** x **in self iff not** x.r.c=0

Now consider the case that C_1 is a Sort, representing the fact that R is a weak entity belonging to S. In this situation, the problem with a C_1-valued object reference in C_2 does not occur, as each C_1-object is necessarily referenced by a C_2-object. Here we may translate Q' to a retrieval method on C_2:

collect y.r **for** y **in self iff not** y.c=0

even though the direction of r is opposite to that of the antijoin. □

4.2.4 Matching Object Graphs

Example 4.5 shows that multiple CODs may be obtained even from a connected QJG. In general, we determine the set of CODs from the subgraph of JMG_Q that results from matching JMG_Q with the QJG, called the *matching object graph* (MOG). It consists of all nodes that represent classes imaging attributes addressed by Q, and edges matching the joins specified by Q. me-edges are added if this results in a higher connectivity of the MOG.

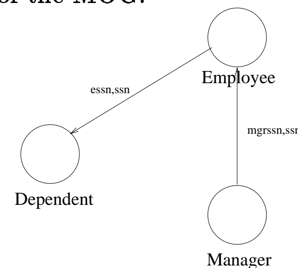


Figure 9: The MOG for the example query

Example 4.6 Figure 9 shows the MOG for the example query of Figure 4. The semijoin

EMPLOYEE$\ltimes_{ssn=mgrssn}$DEPARTMENT has matched the isa-edge from `Manager` to `Employee`, the antijoin from EMPLOYEE to DEPARTMENT has matched the corresponding object-reference. By matching the QJG with the JMG, we have isolated those join conditions of Q that are directly satisfiable in the object schema by travelling the object structure. □

On closer inspection of Figure 9, we note that the distinction between `Manager` and `Employee` is in fact superfluous in the context of Q, since Q considers only those employees that are also a manager, and the semantics of the isa-relation defined in the object schema ensures that these are exactly the instances of class `Manager`. This observation can be generalized to the following proposition (without proof).

Proposition 4.1 : SQL and Inheritance Let an SQL-query Q on a relational schema R contain a join specification R$\bowtie_{A=B}$S that matches an E_{isa}-edge e from a class C_1 to a class C_2 in the JMG of the corresponding object schema O. This join is the SQL-representation of an inheritance mechanism that is implicit in the semantics of the object schema. □

As the inheritance mechanism specified by the SQL-query is implicit in the object-oriented data model, the explicit specification of the superclass is superfluous in the context of a COD. This motivates the merging of super- and subclass nodes in the MOG, as illustrated in the following example.

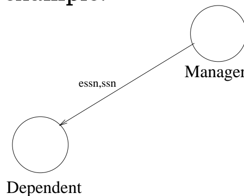


Figure 10: The MOG after eliminating a superfluous superclass

Example 4.7 Figure 10 shows the example MOG after eliminating the superfluous superclass `Employee`. □

Within the MOG, connected subgraphs that form a COD can now be identified. Q is then translated to a set of retrieval methods on the root classes of these COD, and a database method calculating the cartesian product of these partial results.

Example 4.8 The query of Figure 4 describes a single complex object definition with root class `Manager`, as can be seen from Figure 10. We therefore translate the example query of Figure 4 to a method on `Manager`. □

5 Query translation

Having described the derivation of CODs, this section briefly describes the actual query translation. Space limitations prevent us from a full discussion; the interested reader is referred to [12] for details. In the following, we interpret the SQL-**SELECT** as **SELECT DISTINCT** due to the set-oriented nature of our target language.

5.1 Translating a general query

As a SELECT-query Q is a retrieval action on a relational database, it is translated to a database retrieval method in a TM-specification. Assume Q gives rise to a set S of CODs. The database method translating Q calculates the cartesian product of the results of the retrieval methods defined on the root class of each COD, with the addition of conditions spanning multiple CODs (i.e. the unmatched join conditions of Q). If the cardinality of S is 1, Q describes a single complex object definition, and the corresponding database method is a trivial call of the retrieval method on the root class of the COD. In the following, we concentrate on the construction of the TM **collect**-expression that forms the body of a retrieval method M_G of a COD G.

We use the notation Dot(a_i) to stand for the usual dot-notation to express accessing an attribute a_i from a given root class C_R by following a path of object references appearing in the dot-notation. Given a condition c appearing in an SQL-query Q, Dot(c) stands for its equivalent using dot-notation for the attributes involved. Also, *scope*(c) stands for the set of classes imaging attributes involved in c.

5.2 Constructing a collect-expression

The body of M_G is formed by a TM-expression of the form **collect** *selector* **for** x **in self iff** *conds*, where the keyword **self** refers to the extension of the class on which the method is formulated, i.e. C_R. The expression is constructed gradually while travelling the graph-structure of the COD in a depth-first manner, starting at C_R, and guided by the following rules.

Visiting a node When visiting a node v representing a class C for the first time, replace *selector* by *selector* $\cup$ $\{\text{Dot}(a_i) | a_i$ is an attribute of $C; a_i$ is an image of A_i appearing in the selector or an unmatched join condition of $Q\}$. Replace *condition* by *condition* $\bigwedge\{\text{Dot}(c_i) | scope(c_i) \subseteq \{v | v \in V_G; v$ has been visited$\}\}$.

When visiting a node v that has been visited before, do the following: let P be the path from C_R to v

currently explored; let P' be the path from C_R to v that was explored when visiting v for the first time; add $P = P'$ to *conditions*.

Selecting an edge to traverse Let E_v be the set of edges leaving the current node v. The selection of the next $e \in E_v$ to be traversed, is governed primarily by the ordering: 1. Any edge that matched a regular join, 2. Any edge that matched a semijoin, and 3. Any edge that matched an antijoin; and secondarily by the ordering: single-valued object references before multi-valued object references. This assures that outer query blocks of Q are always processed before inner query blocks of Q.

Traversing an edge The consequences of traversing an edge e depend on the following factors:

(1) The type of the object reference r represented by e. r can be single- or set-valued.

(2) The type of the edge $e' \in QJG$ that e matched.

(3) The directions of e and e'.

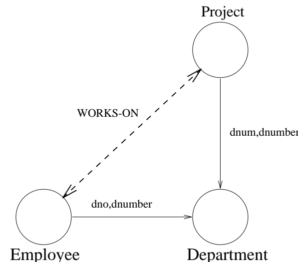
Since we must omit details here, we make some general observations and give an additional query example.

If r is single-valued, traversing r does not give rise to a particular TM-construct. Note that regular join conditions and the SQL-constructs **EXISTS** and **IN** underlying the semijoin conditions are inherently satisfied when traversing a single valued reference that matched this (semi)join.

Traversing a set-valued r gives rise to an extra level of nesting in the **collect** expression. The exact translation of traversing e depends on the type of e'. If e' represents a semijoin or an antijoin, the directions of e and e' play a role, as was illustrated in Examples 4.4 and 4.5.

Example 5.1 Figure 11 shows an example translation on a more involved COD. Notice how traversing a multi-valued object reference gives rise to a nested collect. Note also the condition y.department=x.department, ensuring the equality of both paths to `Department`. □

```
SELECT   name, pname
FROM     EMPLOYEE, WORKS-ON, PROJECT
WHERE    ssn=wssn AND pno=pnumber AND
  EXISTS SELECT *
         FROM DEPARTMENT
         WHERE dnum=dnumber
         AND dno=dnumber AND dname="R&D"
```




```
class retrieval method for Employee
Trans(out ℙ⟨ Name: string Pname : string ⟩)
  unnest
    collect
      collect ⟨ Name= x.name, Pname= y.pname ⟩
      for y in x.projects
      iff y.department = x.department
    for x in self
    iff x.department.dname="R&D"
```

Figure 11: Example query translation

6 Conclusion

In this paper, we argued the feasibility of equipping OO-views of relational databases with methods reflecting operations and constraints from the applications. An algorithm to obtain object methods in an object-oriented DML from SQL-queries has been described, based on the matching of query join graphs with object structures. It was shown that the navigational structure provided by the object model can be exploited to represent joins and nesting constructs contained in an SQL-query.

We have shown that with such a translation, considerable gain in the intuitive meaning of the query can be achieved. It has been shown that the semantics of an object schema can be such that travelling complex object definitions by itself is sufficient to cover for conditions that have to be described in SQL using constructs like **EXISTS** or **IN**. Furthermore, methods translating SQL-queries can be allocated to so-called *root-classes*, which form the focal point of the semantics of the query. As shown in Figures 4 and 5, this may lead to a considerable gain of insight into the meaning of a query.

Although in our examples we used the TM object-oriented DML, the strategy we enunciated is sufficiently general to be applicable for any set-based OODML.

Further research We are currently planning a PROLOG-implementation of the translation procedure, which will be confronted with a test-set of existing queries from large database applications operational within a large international oil company, to

investigate the semantic expressiveness of the results. A later version will use the ODMG query language as its target language.

References

[1] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *Proceedings Seventh European Conference on Object-Oriented Programming, July 26–30, 1993, Kaiserslautern, Germany, LNCS #707*, O. M. Nierstrasz, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1993, 161–184.

[2] M. Castellanos, "A methodology for semantically enriching interoperable databases," in *Advances in Database - BNCOD 11*, Springer-Verlag, New York–Heidelberg–Berlin, 1993, 58–75.

[3] R. H. L. Chiang, T. M. Barron & V. C. Storey, "Reverse engineering of relational databases: Extraction of an EER model from a relational database," *Data & Knowledge Engineering* 12 (March 1994), 107–142.

[4] U. Dayal, "Query processing in a multidatabase system," in *Query Processing in Database Systems*, W. Kim, D. S. Reiner & D. S. Batory, eds., Topics in Information Systems, Springer-Verlag, New York–Heidelberg–Berlin, 1985, 81–108.

[5] U. Dayal, "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," in *Proceedings of Thirteenth International Conference on Very Large Data Bases, Brighton, England, September 1–4, 1987*, P. M. Stocker, W. Kent & P. Hammersley, eds., Morgan Kaufmann Publishers, Los Altos, CA, 1987, 197–207.

[6] J. Flokstra, Maurice van Keulen & J. Skowronek, "The IMPRESS DDT: A database design toolbox based on a formal specification language," in *Proceedings ACM-SIGMOD 1994 International Conference on Management of Data*, ACM Press, New York, NY, 1994, 506.

[7] P. Johannesson, "A method for transforming relational schemas into conceptual schemas," in *Proceedings Tenth International Conference on Data Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1994, 190–201.

[8] W. Kim, "On optimizing an SQL-like nested query," *ACM Transactions on Database Systems* 7 (September 1982), 443–469.

[9] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham & S. Dao, "Construction of a relational front-end for object oriented database systems," in *Proceedings Ninth International Conference on Data Engineering, Vienna, Austria, April 19–23, 1993*, IEEE Computer Society Press, Washington, DC, 1993, 476–483.

[10] A. P. Sheth & J. A. Larson, "Federated database systems for managing distributed, heterogeneous and autonomous databases," *ACM Computing Surveys* 22 (September 1990), 183–236.

[11] S. D. Urban & T. ben Abdellatif, "An object-oriented query language interface to relational databases in a multidatabase environment," in *Fourteenth International Conference on Distributed Computing Systems*, 1994, 287–295.

[12] M. W. W. Vermeer & P. M. G. Apers, *Enhancing the semantics of federated schemata by translating SQL-queries into object methods*, Universiteit Twente, Enschede, The Netherlands, 1994.