[Kim 89] Kim W., Ballou N., Chou H.-T., Garza J.F., Woelk D.: *'Features of the ORION Object-Oriented Database System'*, chapter 11 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. 251-282

[KL 92] Keim D. A., Lum V.: *'Visual Query Specification in a Multimedia Database System'*, Proc. Conf. Visualization, CS Press, Los Alamitos, CA., 1992.

[Loh 91] Lohman G.M., Lindsay B., Pirahesh H., Schiefer K.B.: *'Extensions to Starburst: Objects, Types, Functions and Rules'*, Comm. of the ACM, Vol. 34, No. 10, 1991, pp. 94-109.

[Mel 92] Melton J. (ed.): *'Database Language SQL (SQL3)'*, ISO/ANSI working draft, X3H2-92-055 DBL CNB-003, July 1992.

[MM 90] Markowitz V., Makowsky J.: *'Identifying Extended Entity-Relationship Object Structures in Relational Schemas'*, IEEE Tran. on Software Engineering, Vol. 16, No. 8, 1990.

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

[RC 88] Rogers T. R. , Cattell R. G. G.: *' Entity-Relationship Database User Interfaces'*, in: Readings in Database Systems, M. Stonebraker (ed.), 1988.

[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.

[Shi 81] Shipman D.W.: *'The Functional Data Model and the Data Language DAPLEX'*, ACM Trans. on Database Systems, Vol. 6, 1981, pp. 140-173.

[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.

[Sto 93] Stonebraker M.: *'The Miro DBMS'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., 1993, pp. 439.

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

[VAO 93] Vadaparty K., Aslandogan Y. A., Ozsoyoglu G.: *'Towards a Unified Visual Database Access'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., 1993, pp. 357-366.

[Cas 93] Castellanos M.: *'Semantic Enrichment of Interoperable Databases'*, Proc. 3rd Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria, 1993, pp. 126-129.

[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.

[Haa 89] Haas L.M., Freytag J.C., Lohman G.M., Pirahesh H.: *'Extensible Query Processing in Starburst'*, Proc. ACM-SIGMOD Int. Conf. on Management of Data, 1989, pp. 377-388.

[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 88] Heuer A.: *'An Object Algebra and its Connection to the NF$^2$- and Flat Relational Algebra'*, Proc. Workshop on Relational Databases and their Extensions, 1988, in: Informatik-Bericht, Vol. 4, Institut für Informatik, TU Clausthal, 1988.

[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.

[HK 93] Hohenstein U., Körner C.: *'Object-Oriented Access to Relational Database Systems'*, Proc. GI-Fachtagung, Braunschweig, 1993, in: Datenbanksysteme in Büro, Technik und Wissenschaft, Springer, 1993, pp. 246-255.

[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'*, to appear in: Proc. 4th. Int. Conf. on Database and Expert Systems Applications (DEXA'93), Prague, Czechoslovakia, 1993.

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 intend to use our query translation algorithm as a basis for an advanced integration of relational systems into a heterogeneous multidatabase system.

## References

[Agr 90] Agrawal R. et al.: *'OdeView: The Graphical Interface to Ode'*, Proc. ACM-SIGMOD Int. Conf. on Management of Data, Atlantic City, 1990.

[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.

[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.

- $project(a\_new_1=a\_old_1, ..., a\_new_l=a\_old_l)$: $Res \rightarrow Res$
  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:

$$f_Q = f_0 \circ f_1 \circ \cdots \circ f_{(n-1)} \circ f_n \quad \text{with } f_0 \text{ being the } \varepsilon\text{–function.}$$

The formatting primitives and their concatenation to the formatting function $f_Q$ are illustrated in figure 5 using query example 2.

## 5 Summary and Conclusions

Relational database systems are widely used in research and industry. A major problem of relational systems are the poor query facilities of SQL. In this paper, we described basic algorithms which enhance the functionality and usability of existing relational databases and allow to query them like object-oriented databases. The main contribution of this paper is the query translation algorithm which 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 adequate joins and subqueries on the relational side, the conditions are replaced by equivalent SQL conditions and, since SQL can not 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 to reconstruct 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, systems built on the basis of our schema transformation and query translation algorithms may be commercially available in a short time. The implementation of the schema transformation and operation translation

$f_0$ =ε

$f_1$ = *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 5: Formatting Function for Query Example 2**

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* → *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* → *Res*
groups *Res* according to equal values of *by(attr_list)* and for each group, the attributes listed in *op(attr_list)* are combined to sets if *op* = *nest*, or aggregated using the corresponding aggregate operator if *op* = *count, avg, sum, min, max.* 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).*

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 4, 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 are grouped into a set. Now, the first structs ‘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’ $\equiv>$ $\{V_1.\text{name} \mid V_1 \in \text{D.passengers}\}$ = ‘Jones’

$\equiv>$ $\exists V_1: V_1 \in \text{D.passengers} \wedge V_1.\text{name}$ = ‘Jones’

Only applying the resolution according to the generalization of dot-notation to $\{V_1.\text{name} \mid V_1 \in \text{D.passengers}\}$ = ‘Jones’ results in $\{V_1.\text{name}$ = ‘Jones’ $\mid V_1 \in \text{D.passengers}\}$ 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.

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.

It is interesting to note, that after step 1 of the translation process the query is in a form that can be easily translated into other object-oriented languages like $O_2$SQL [BCD 92]. In query example 2, the partially resolved set-notation of the query can also be written as

```
{((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 $\wedge$ P.address like ‘%8000 München%’}
```

which is equivalent to the following $O_2$SQL query

```
select tuple (p: tuple (pn: P.name, pa: P.address),
           d: tuple (d1: (select tuple (di: $V_1$.did,
                                  dn: (select $V_2$.name from $V_2$ in $V_1$.passengers))
                        from $V_1$ in P.departures),
                    d2: count(P.departures) ))
from P in Passenger where P.address like ‘*8000 München*’.
```

## 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 con-

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 $V_2$, Pass_Dept $V_3$, Pass_Dept $V_4$
where P.pid = $V_3$.pid and $V_3$.did = $V_1$.did and $V_1$.did = $V_4$.did and $V_4$.pid = $V_2$.pid and P.address like '%München%

**Figure 4: Translation of Query Example 2**

from inner nesting levels need to be removed (c.f. translation step 4b in figure 4). Formally, the flattening of one nesting level can be described as:

$$\{(x, \{y \mid y \in Y \wedge p(x, y)\}) \mid x \in X \wedge q(x,)\}$$
$$\equiv> \{(x, key(x), y) \mid y \in Y \wedge p(x, y) \wedge x \in X \wedge q(x)\}.$$

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 4). Again, in this step the formatting function is extended by the inverse operations and key

**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 $V.m_1. \ldots .m_n$ with $m_n$ being an aggregate operation into $\{v \mid v \in V.m_1. \ldots .m_{n-1}\}.m_n$. Then all object references in $V.m_1. \ldots .m_{n-1}$ can be resolved by join predicates as described above. In some cases, however, no additional joins may have to 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 4). 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 fined 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 3). 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 [redacted] are also resolved successively as [redacted] if at least one of the $m_{i_1}$ is directly applicable to *V* and as [redacted] 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

$\{(\text{P.name}, \text{CP.name}) \mid \text{P} \in \text{Passenger} \wedge \text{CP} \in$
$\{\text{V}_1.\text{passengers} \mid \text{V}_1 \in \text{P.departures}\}\}$ is translated into

$\{(\text{P.name}, \text{V}_1.\text{passengers.name}) \mid \text{P} \in \text{Passenger} \wedge \text{V}_1 \in \text{P.departures}\}$

$\equiv_{(\text{step 1})} \{(\text{P.name}, \{\text{V}_2.\text{name} \mid \text{V}_2 \in \text{V}_1.\text{passengers}\}) \mid \text{P} \in \text{Passengers} \wedge \text{V}_1 \in \text{P.departures}\}$.

More formally, the translation can be expressed as

$\{(x, y) \mid x \in X \wedge y \in \{h(z) \mid z \in Z \wedge p(x, z)\} \wedge q(x, y)\} \Rightarrow \{(x, h(z)) \mid x \in X \wedge z \in Z \wedge p(x, z) \wedge q(x, h(z))\}$ with a subsequent resolution of generalized dot-notation (c.f. step 1).

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

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

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

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

$\cong$
select P.name, P.address
from Passenger P, Departure D, Airline $V_1$
where join(P, D) and join(D, $V_1$) and D.start = ‘06/18/93’ and $V_1$.name = ‘Lufthansa’

$\cong$
select P.name, P.address
from Passenger P, Departure D, Airline $V_1$, Pass_Dept $V_2$
where P.pid = $V_2$.pid and $V_2$.did = D.did and D.airline-id = $V_1$.airline-id and D.start = ‘06/18/93’ and

**Figure 3: Translation of Query Example 1**

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 occuring subqueries are recursively transformed into corresponding set expressions. In figures 3 and 4, 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 $V.m_1. \dots .m_n$ in the ‘select’ or ‘for each’ clause are successively resolved as $(V.m_1). \dots .m_n$ if $m_1$ is a method defined for $V$ and as $\{v.m_1 | v \in V\}.m_2. \dots .m_n$ if $V$ is set-valued and $m_1$ is not de-

Let $R_i := \mathit{flat\_type}(V.m_1.m_2. \ldots .m_i)$ 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:

Example:

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 $V_i$.

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:

$$f_Q(res(\text{S}, \text{RDB})) = res(\text{Q}, \text{ODB}), \qquad (*)$$

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*(S’, RDB) $)$ = *res*(Q, 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:

the structured result {(Smith, {401, 403}), (Smith, {401})}. A corresponding SQL query together with its result is also given in figure 2. 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.

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

**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 2: Instances of the Relational and the Virtual Object-Oriented Database**

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 2 against ODB yields 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. Additionally, 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 2 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 object class from another without explicitly joining them. It is some kind 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 to define the result structure by square brackets. The generalization of the dot-notation to sets is an intuitive but powerful continuation 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 1. 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 '8000 München'.

**Example 2:**
```
select  P.[name, address], P.departures.[[did,
                          passengers.name], count]
for each  Passenger P
where  P.address like '%8000 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

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) 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).

## 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 ($O_2$SQL [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, automatically a new (temporary) object class is 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 allows to directly access one

```
FlightDB:
    Passenger (pid: Integer; name: String; address: String)
    Departure (did: Integer; start: Date; flight: Integer; airline-id: String;
               plane-id: Integer)
    Pass_Dept (did: Integer; pid: Integer; booking: Date)
    Airline(airline-id: String; name: String)
    Plane(serial-nr: 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 1: Example for the Schema Transformation**

The basic steps of the schema transformation algorithm are: 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 1, an example for a relational database FlightDB 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 [KKM93a].

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 deducted 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 both of 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.

tems will play an important role in future commercial database systems, we believe that for many practical environments it is important not to change the relational systems with their large existing databases and application programs. Most database vendors offer gateways that provide some kind of cross-database access [Syb 90, Ing 92, Ora 92] allowing to use specific new database systems in conjunction with existing relational ones. The main purpose of gateways, however, is to allow databases to work together and not to enhance relational systems. As a result, most gateways only offer a limited functionality, providing at most the query facilities that are provided by the relational system itself. In the area of multidatabase systems, again the main goal is to allow database systems to work together. Most research done in this area has been focussing on schema integration and transparent inter-database access but only few researchers address the issue of enhancing the functionality of relational systems that are part of the federation. Some papers address the issue of schema enrichment [MM 90, Cas 93] but little work has been done on query translations that allow an enhanced querying of existing relational databases [Heu 88, RPR 89].

The goal of our query algorithm is to enhance the querying of existing relational databases allowing the querying process to be easier and more intuitive. This can be achieved by using our object-oriented query language SOQL [KKM 93b] which is an extension of SQL allowing queries to be more orthogonal, results to be structured, and uses direct object references instead of explicit joins. We found that there is a quite large class of SOQL queries that can be automatically translated into equivalent SQL queries. A prerequisite is that information about the schema transformation which has been performed beforehand is available. Although not all queries that can be expressed in SOQL can be efficiently translated into SQL queries (examples are queries with conditions that involve set comparison or user-defined functions, c.f. section 3), the effort of translating queries from SOQL to SQL is justified since SOQL queries are much shorter, easier to write and understand and more intuitive than corresponding SQL queries.

Multidatabase or interoperable database systems are aimed at facilitating the use of new database technology in real world environments. They try to provide a framework for the smooth co-existence of legal and new database systems by allowing an integrated and transparent access [SL 90] but, in general, they do not improve the querying of existing databases. Important tasks in building a multidatabase system are resolving schematic discrepancies, transforming and integrating the schemas, decomposing and translating queries and combing the results. For improving the query interface to existing relational database systems, a simple transformation and integration of the schemas is not sufficient. The schemas need to be enriched semantically which, in most cases, is only possible with additional information acquired from the user. For industrial environments with quasi static schemas, schema enrichment, transformation and integration may be done once in the beginning as a user guided process and need only to be repeated if new databases join the federation or if schema changes occur. For the tasks that are related to query processing, user interaction is not feasible since they have to be done each time, a query is processed by the system.

The query translation algorithm which will be presented in this paper is important not only in the context of multidatabase systems with an object-oriented global common data model but in any system that provides object-oriented access to existing relational databases. One possibility, for example, is an object-oriented front-end to a relational system similar to the one presented in [KKM 93b] which allows an object-oriented querying of existing relational databases without migrating or transforming data and changing existing application programs. Our query translation algorithm may also prove useful to allow future object-oriented extensions of SQL (e.g. SQL3 [Mel 92]) to work on existing SQL2 databases.

The idea of providing object-oriented access to relational databases is not new but most researchers have been working on extending existing database systems or providing cross-database access [Loh 91, Mel 92, HK 93, Sto 93]. Although object-oriented extensions of relational sys-

## 1 Introduction

Relational database systems are widely used in research and industry. For traditional application areas like accounting, reservation systems, etc., the relational data model seems to be adequate providing suitable modeling and performance characteristics. The main reasons for using the relational data model are: It is simple, well known and has a firm theoretical basis. Since relational database systems are used by far most commonly in real world applications, their dominance will remain in the near future. However, relational databases are not adequate for applications such as CAD, CAM, CIM, CASE or Multimedia which require more functionality, especially better modeling capabilities and more expressive query languages. A lot of research has been going on over the last decade to improve the limited capabilities of the relational model to express semantic aspects, i.e. relationships, structured entities and procedural aspects. The result has provided major advances in database technology, e.g. the object-oriented and extended relational database systems with their extended semantic modeling capabilities (e.g. [Shi 81, Kim 89, Loh 91, BDK 92, Sto 93]), advanced query languages (e.g. [Haa 89, BCD 92]) and graphical user interfaces (e.g. [RC 88, Agr 90, KL 92, VAO 93]).

Using advanced database technology for improving access to existing databases usually requires a complete system change or migration making it necessary to convert the existing databases with all their application programs that have been written and successfully used over the years. A further difficulty in the migration process is that, in general, the relational 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, in most cases system changes or migrations are quite expensive, but even worse is the small chance of a complete system failure. Companies therefore limit system changes to the absolute minimum and, if changes are unavoidable, they are planned carefully well in advance.

# Query Translation of an Object-Oriented into a Relational Query Language

Daniel A. Keim, Hans-Peter Kriegel, Andreas Miethsam

Institute for Computer Science, University of Munich
Leopoldstr. 11B, D-80802 Munich, Germany
{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 an improved query interface for existing relational database systems. The translation algorithm, we propose in this paper may be used to directly access relational databases, but it may also be useful in the context of object-oriented multidatabase systems to translate the common global query language into the query languages of participating relational databases. Necessary steps in providing object-oriented access to relational databases are schema enrichment and transformation as well as query translation. The main focus of this paper is the query translation which has to be performed fully automatically since it has to be done each time, a query is processed by the system, whereas schema enrichment and transformation may be done only once in the beginning. Our query translation algorithm ensures a full 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. Problems may occur if additional user defined functions are used.

***Keywords:*** *query translation, query languages, relational database systems, object-oriented database systems, multidatabase systems, schema enrichment and transformation*

# Query Translation of an Object-Oriented into a Relational Query Language

*Daniel A. Keim*
*Hans-Peter Kriegel*
*Andreas Miethsam*

**Ludwig-Maximilians-Universität München**
**Institut für Informatik**

**Bericht 9325**
**Dezember 1993**