Transforming Queries from a Relational Schema to an Object Schema: A Prototype Based on F-logic*

Yahui Chang and Louiqa Raschid and Bonnie Dorr

Department of Computer Science, University of Maryland, College Park, MD 20742

Abstract. This paper describes a technique to support interoperable query processing when multiple heterogeneous databases are accessed. The problem is to support query transformation transparently, so a user can pose queries locally, without any need of global knowledge about different data models and schema. In this paper, we focus on transforming SQL source queries, posed against a relational schema, to XSQL queries to be evaluated against an object schema which represents sharable information. We will describe the extraction algorithm which extracts semantics from a source SQL query, and some of example mapping rules which perform query transformation by accessing mapping knowledge of the models, schema and query languages. The mapping knowledge is represented in a canonical form, using a second-order logic representation, namely F-logic.

1 Introduction

One of the most important requirements of a heterogeneous information system is to support interoperable query processing when multiple heterogeneous servers are accessed. The problem is to support query transformation transparently, so a user can pose queries locally, without needing global knowledge about different data models and schema. After such a transformation, a query may be answered using information from multiple sources. In our research described in this paper and in (Raschid *et al.* (1993, 1994)), we present a technique to achieve interoperability, through capturing knowledge about source and target query languages, data models, schema and mapping information, in local and mapping knowledge dictionaries, and applying a set of mapping rules to obtain a transformation. We represent the dictionary knowledge and mapping rules using a second-ordered logic language, F-logic (see Kifer and Lausen (1989)). F-logic has appropriate modeling constructs for representing mapping knowledge, it can represent object schema fairly concisely, and it has an interpreter for our programs (Lawley (1993)).

This research can be placed in the context of other research in the area of *representational heterogeneity*, where sharable knowledge is stored in multiple

* This research has been partially supported by the Defense Advanced Research Project Agency unde r grant DARPA/ONR grant 92-J1929.

schema using different models, leading to possible mismatch between schemas and query languages. Many approaches advocate the use of a global schema (Ahmed *et al.* (1993), Arens *et al.* (1992), Kim *et al.* (1993)). A disadvantage of the global schema approach is that a single global schema is needed to be accepted by all users; this may not be feasible in some environments. For example, queries posed against the local database (legacy applications) may wish to access information in other servers, without any change to be adapted to the new global schema. To meet this need, our approach does not require schema integration among the multiple servers. Moreover, our approach addresses the problem of transforming queries transparently, so the local user (or legacy application) does not need explicit knowledge (about the global schema or mapping knowledge among the schema) to pose a query. The research that is closest to our work is described by Lefebvre *et al.* (1992) and Qian (1993), but their approach is limited to one single model. In the companion paper (Raschid *et al.* (1994)), we have described in details these research approaches and compared them with our research.

Our approach encapsulates the information on resolving representational heterogeneity in a knowledge dictionary and uses this to support query transformation. Users need not write their queries against a global schema; thus, we provide interoperability in a transparent manner. In addition, the knowledge dictionary represents the mapping and transformation information in a declarative manner; this can also be treated as a knowledge source, to be used for providing explanations, etc. Finally, our approach also extracts relevant semantics from a source query; this allows the transformed queries to reflect the user's requirements more closely and may be more efficient to process. In Raschid *et al.* (1993, 1994), we studied the problem of transforming queries posed against an object schema into queries against a relational schema. We considered XSQL queries, an extension to an SQL query allowing *path expressions* that represent a path in an object schema, along a class hierarchy or along reference pointers.[2] In this paper, we address the issue of transforming SQL queries posed against a relational schema to produce XSQL queries against the object schema.

This paper is organized as follows: section 2 gives examples of transforming from SQL queries to equivalent XSQL queries. Section 3 describes the extraction algorithm which extracts semantics from a source SQL query and then represents the semantics in a canonical form. Section 4 describes the F-logic implementation which transforms a canonical form for an SQL query to an equivalent XSQL query. The extraction algorithm has been implemented in Lex, Yacc and C. The mapping rules have been implemented using the F-logic interpreter (Lawley, 1993).

2 Examples of Transforming Queries from Relational to Object Schema

[2] The detailed syntax of XSQL queries is described in Kifer *et al.* (1992).

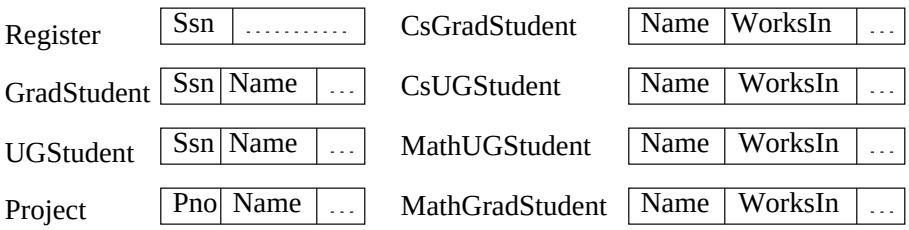

Fig. 1. Example relational schema

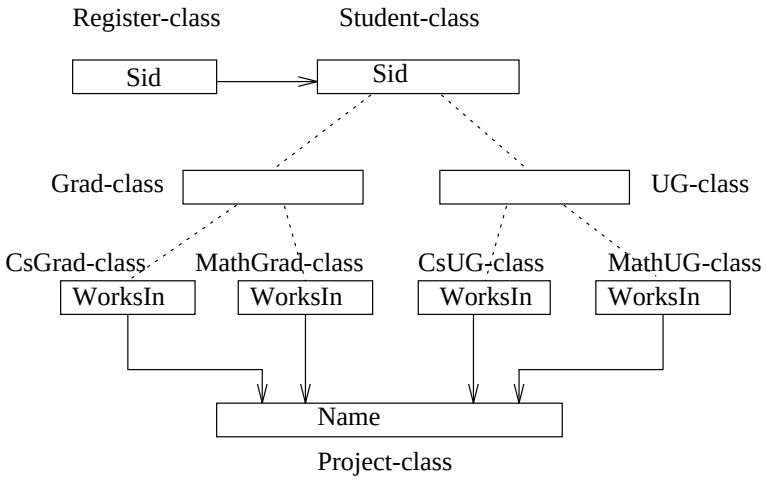

Fig. 2. Example object oriented schema #1

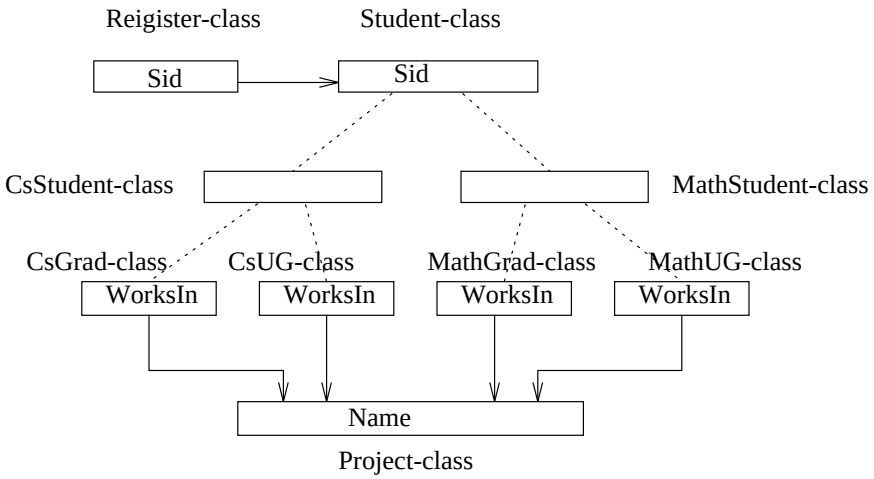

Fig. 3. Example object oriented schema #2

We use a sample relational schema (Figure 1) and two object schemas (Figures 2 and 3). In the relational schema, relation Register has an attribute Ssn which refers to the primary identifier, social security number, for a student who is currently registered. There are several relations containing information about students. GradStudent and UGStudent have Ssn as the primary key. Four other relations correspond to graduate and undergraduate students in two departments; these use Name as the primary key and have an attribute WorksIn which refers to a Project in which the student participates. The relation Project describes these projects and is identified based on Pno. The attribute PI denotes the principal investigator.

In the object schema,[3] each node is an object. The dotted line represents a class hierarchy, and a solid arc represents a pointer to another object, e.g., the attribute Sid of an object Register-class points to an object Student-class. In object schema #1, Student-class has two immediate sub-classes Grad-class and UG-class. Each of them has also two sub-classes, identifying the Cs and Math graduate and undergraduate students (4 classes). In contrast, in object schema #2, CsStudent-class and MathStudent-class are the sub-classes of Student-class. Each of these classes has two sub-classes, for graduate and undergraduate students, respectively. To compare the two object schemas, in schema #1, all grad students are in one class Grad-class but in schema #2, they are in two classes, CsGrad-class and MathGrad-class.

Queries against the relational schema will be expressed using SQL. To express a query against the object schema, we use a XSQL-like query (see Kifer *et al.* (1992)). We use simple XSQL queries that only differ from SQL queries in the path expression of the where clause. The *path expression* we consider is of the following form:

$$\mathrm{sel}_0.\mathrm{Att}_1\{[\mathrm{sel}_1]\}.\cdots.\mathrm{Att}_m\{[\mathrm{sel}_m]\}$$

A selector sel_i is a class label and Att_i is an attribute. The attribute Att_{i+1} following each selector sel_i is either directly defined in the class sel_i or it is an inherited attribute for sel_i and must be defined for one of the super-classes of sel_i. Each $sel_i.Att_{i+1}$ can either return a value or a pointer to an object. Braces $\{\}$ denote optional terms of the path expression. The selector sel_{i+1} following any expression such as $sel_i.Att_{i+1}$ indicates that sel_{i+1} could correspond to the class object that is pointed to, or it could be any sub-class of this class.

Suppose a user wants to identify all principal investigators of the projects in which currently registered computer science graduate students conduct research. This would correspond to the following SQL query against the relational schema:

Query 1 in relational schema
Select PI
from Register, GradStudent, CsGradStudent, Project
where Register.Ssn = GradStudent.Ssn and
GradStudent.Name = CsGradStudent.Name and
CsGradStudent.WorksIn = Project.Pno

[3] Each object in the schema is a *class* object which may be distinguished from the *individual* objects or instances.

Note that tuples from the relation Register cannot be joined with tuples from the relation CsGradStudent directly, because they use different attributes as keys. In the object schema, these joins must be interpreted as selecting those students in CsGrad-class, and the projects in which they work. CsGrad-class occurs as a sub-class of different classes in object schema #1 and #2. However, the equivalent XSQL query for these object schemas will be the same, since the XSQL query need not specify the different paths from Student-class to CsGrad-class in the two schema due to the fact that CsGrad-Class is a sub-class of Student-class in both cases. The query is as follows:

Query 2 in object schema #1 and #2:
Select Z.PI
from Register-class X, CsGrad-class Y, Project-class Z
where X.Sid[Y].WorksIn[Z]

3 Extraction Algorithm

We will present the extraction algorithm in this section. The algorithm operates on a source relational query and produces a corresponding canonical structure CRrel. There are several reasons for adopting such a structure instead of direct translation of a source query to the target query language form. First, different query languages (object or relational) have different constructs, e.g., path expression in XSQL, functional composition in OSQL, etc. In addition, we have the heterogeneity of the different schema themselves. Using a canonical structure allows us to separate the mapping knowledge due to the heterogeneity of schema from the knowledge needed to translate between the constructs in the different languages. If we use a direct translation approach, the rules will be very complicated because they have to resolve the schema conflicts and also consider the syntax of each language. By extracting the semantics of the query and representing them in a canonical form, the semantics is considered independent of the query languages; thus, we simplify the mapping rules. This is precisely analogous to arguments for the use of an interlingual representation over a direct-mapping approach in the field of natural language translation (see, e.g., Dorr (1993)).

A second reason for adopting a canonical structure is that the user might provide a poor query (e.g., redundant joining pairs in the WHERE part of a SQL query) because of lack of sufficient knowledge about the local schema. The EM has access to the local schema dictionary and may be able to modify the query to provide an optimal form for query transformation. This might be especially useful when the mapping information is incomplete. By using a canonical form, we can identify the exact mapping information we need.

We use a data structure, CRrel, to represent the semantics of an SQL query. Here we only discuss the constructs corresponding to the WHERE clause. The WHERE clause of an SQL query is used to express constraints and probably involves complicated operations. We assume the WHERE clause only consists of conjunctions of the form: cond_1 AND cond_2 AND $\cdots$ cond_n. Each conjunct cond_i could be viewed as a predicate that is represented by comparing the attribute of relations with some other values, set-operation involving nested queries, etc.

The CRrel structure is represented as an F-logic data structure. Details of F-logic syntax are given in the next section.

CRrel[sub-rel $\leftrightarrow$ (R_{sup}, R_{sub});
reference $\leftrightarrow$ REF;
comparison $\leftrightarrow$ (REL-OPR, OP1, OP2);
set-operation $\leftrightarrow$ (SET-OPR, OP1, OP2)]

The values of the attribute *sub-rel* is a set of pairs with the form (R_{sup},R_{sub}), which means the key of the relation R_{sub} is a subset of the key of the relation R_{sup}. The value of the attribute *reference*, REF, is a set of ordered lists of quadruples. Each quadruple has the form (R1,A1,R2,A2). This corresponds to the joining pair R1.A1 = R2.A2 in SQL when there is a foreign key relationship between R1.A1 and R2.A2. The value of the attribute *comparison* is a set of triples with the form (REL-OPR, OP1, OP2), where REL-OPR represents a relational operator, OP1 is the first operand and OP2 is the second operand. This corresponds to a comparison of the values of OP1 and OP2, and does not have special structural meanings. The last attribute *set-operation* corresponds to the operation involving sets. SET-OPR could be IN, or NOT IN, etc., which are used to test if the value of OP1 is IN or NOT IN the set represented by OP2. OP2 is a nested query and will be represented by another CRrel structure.

We now describe the portion of the algorithm for the EM which operates on a set of joining relation-attribute pairs. The aim is to consider each joining pair in the WHERE clause, and use schema information such as keys, key inclusion dependency, foreign keys, etc., to simplify the query, eliminate redundant pairs, and obtain relevant structure information.

Extraction Algorithm

- Input : a set of joining relation-attribute pairs
- Output : CRrel in F-logic form

1. Build a graph where each node corresponds to a relation, and each edge is either an *inc-link* or a *ref-link* when applicable. The *inc-links* represent a key inclusion dependency, and the *ref-links* represent a foreign key relationship.
   For each input pair (R1.A1 = R2.A2), create new nodes for relations R1 or R2 if they have not been created previously. Then check if the inclusion dependency exists and create corresponding *inc-links* and *ref-links*:
   (a) If A1 and A2 are both keys, check if there exists a key inclusion dependency between them. If so, create an inc-link pointing from the node corresponding to the subset relation to the superset relation. For example, if (R1.A1) $\subseteq$ (R2.A2), the link is pointing from node R1 to node R2.
   (b) If (R1.A1) is a foreign key of R2 which has the key A2, create a ref-link pointing from the node R1 to the node R2. Annotate the link with the quadruple (R1, A1, R2, A2) to denote the joining relation-attribute pairs. Also test if R2.A2 is a foreign key of relation A1.

(c) If the pairs do not have specific structural meanings, output the F-logic form: CRrel[comparison $\leftrightarrow$ ("=",R1.A1,R2.A2)]

2. Check inc-links and build *inclusive classes*.
   Let each node which does not have any incoming inc-links in the graph be the root, say R1. This represents the most specific subset in that hierarchy with respect to inclusion dependency. Traverse through the inc-links and include each visited node, say R_j, in the inclusive class denoted by R_i. Note that R_j may be in several inclusive classes if it has more than one incoming inc-links. The corresponding output structure is:
   $$\text{CRrel[sub-rel} \leftrightarrow (R_j,\ R_i)].$$
   If there is a cycle through the inc-links, all those corresponding relations $(R1, \cdots Rn)$ will have the same set of values for their keys. This means that they actually correspond to a vertically partitioned relation where the attributes of a tuple are distributed in several relations. We will randomly pick one relation R_i as the name of the inclusive class and create the following constructs:
   CRrel[sub-rel $\leftrightarrow$ (R1, R_i)], $\cdots$, CRrel[sub-rel $\leftrightarrow$ (R_i, R_i)], $\cdots$, CRrel[sub-rel $\leftrightarrow$ (Rn, R_i)].
   For those nodes R_i which do not have incoming and outgoing inc-links, we will produce the following structure: CRrel[sub-rel $\leftrightarrow$ (R_i, R_i)].
3. Check the ref-links and produce ordered lists.
   (a) This step is to modify the graph by removing the inc-links and "collapse" the nodes which are in the same inclusive classes.
   For each node, say R, whose inclusive class, say C, is not equal to R, do the following:
   for each outgoing ref-link n pointing from node R to a certain node R', add a new ref-link pointing from C to the inclusive class C' of R', and delete n;
   for each incoming ref-link n pointing to R from another node R', add a new ref-link pointing to C from the inclusive class of R', and delete n.
   (b) This step is to navigate through the ref-links and produce an ordered list which corresponds to a sequence of referential relationships.
   Let each node which does not have incoming ref-links be a root. Traverse through the links from the root to each leaf to create an ordered list of joining pairs. Suppose there is a link pointing to the node N, with annotation (R1,A1,R2,A2), and a link pointing from the node N, with annotation (R3,A3,R4,A4), then output:
   CRrel[reference $\leftrightarrow$ cons(cons(L,(R1,A1,R2,A2)),(R3,A3,R4,A4))],
   where L is the output from the previous traversal. Initially it is given the value nil.[4]

[4] For notational simplicity, we will replace the expression cons(nil,(R1,A1,R2,A2)) by just (R1,A1,R2,A2).

4 Representation for Mappings

This section discusses the mapping rules, which utilize the HTrel-objMapping information from the mapping knowledge dictionary. These rules are represented as a F-logic program. We will give a brief description of the syntax of F-logic in section 4.1, explain the data structures used by the mapping rules in section 4.2, and present some examples to describe the functionality of the mapping rules in section 4.3.

4.1 F-logic

We borrow the brief description from Lefebvre *et al.* (1992) to introduce the syntax of F-logic. In F-logic, the *instance term* $o : c$ means that the object o is an instance of class c. A *data term* $o[m@a_1, \cdots, a_n \rightarrow v; m'@a_1, \cdots, a_p \leftrightarrow \{v', v''\}]$ means that the value of the functional method m with arguments a_1 to a_n for the object o is a set containing the values v' and v''. If a method m has no arguments, "@" will be omitted. The symbol $\rightarrow$ indicates a single-valued method, and the symbol $\leftrightarrow$ indicates a set-valued method.[5] An object can be denoted by a constant, or a term. For example, $dept(cs)$ is a valid object identifier. Notational conventions allow us to write $o[m' \leftrightarrow v]$ instead of $o[m' \leftrightarrow \{v\}]$ for a single element of a set-valued method; the expression $o[m \rightarrow v; n \rightarrow v']$ is equivalent to $o[m \rightarrow v] \wedge o[n \rightarrow v']$.

A F-logic program consists of a set of data declarations(data or instance terms), and a set of *deduction rules*. A deduction rule has a *head*, which is a data term, and a *body*, which is a conjunction of data and instance terms. Disjunction and negation are allowed in the body of rules.

4.2 Data Structures

There are three kinds of data structures used by the mapping rules described in the section 4.3. CRrel produced by the Extractor Module represents the semantics extracted from an SQL query. This was described in the previous section. HTrel-objMapping captures the mapping information from a relational schema to an object schema D. The class hierarchy in D is represented by the method *super-class*. Applying the mapping rules and HTrel-objMapping,we produce TQobj which represents the transformed query in the object schema. Its method *sub-class* will return the corresponding class label for the inclusive class of R. The method *path* will produce an XSQL path expression, given an ordered list of joining pairs. The method *join* is invoked to produce an explicit join in XSQL format.

HTrel-objMapping(R, D)[mapped-class $\leftrightarrow$ C[super-class $\leftrightarrow$ C'];
mappings $\leftrightarrow$ map(R,A,D)[object-attr-domain $\leftrightarrow$ (O,AN,OO)]]

TQobj(CRrel, D)[sub-class@R $\rightarrow$ O; path@J $\rightarrow$ L; join@J $\rightarrow$ K]

[5] This symbol is different from the one used in the original paper.

4.3 Mapping Rules

The mapping rules can be classified into three groups. The first group *Ident-Class-Hier* deals with *inclusive classes*. The second group *Object-Ptr* determine the appropriate path in the object schema corresponding to a given ordered list of joining pairs in the relational schema. The last group *Exp-Join* is used when an explicit join is needed. We now show some examples of the application of the rules. The object schema #2 in Figure 2 will be called d_2.

- **Group One: Ident-Class-Hier**
  The rules in this group determines the class name in the object schema corresponding to the inclusive class associated with a relation. We must check if the class hierarchy in the target object schema corresponds exactly to the inclusive classes in the relational schema. If so, we produce an optimal query for the relevant (most specific) class; otherwise, we generate an explicit join.
  Suppose the relation GradStudent is in the inclusive class denoted by Cs-GradStudent and the corresponding CRrel is: CRrel[sub-rel $\leftrightarrow$ (GradStudent, CsGradStudent)]
  Also, CsGradStudent maps to the class CsGrad-class, and CsGrad-class is a subclass of Grad-class. Thus, the inclusive class hierarchy of the relation schema corresponds exactly to the subclass hierarchy in the object schema d_2. The instantiation of the mapping rules will be as follows:

  TQobj(CRrel,d_2)[sub-class@GradStudent $\rightarrow$ CsGrad-class] $\leftarrow$
  CRrel[sub-rel $\leftrightarrow$ (GradStudent, CsGradStudent)] $\wedge$
  HTrel-objMapping(GradStudent,d_2)[mapped-class $\leftrightarrow$ Grad-class] $\wedge$
  HTrel-objMapping(CsGradStudent,d_2)[mapped-class $\leftrightarrow$ CsGrad-class
  [super-class $\leftrightarrow$ Grad-Class]]

- **Group Two: Object-Ptr**
  The rules in this group produce an XSQL-like path expression. The parameter of the method *path* of TQobj is an ordered list of quadruples. A quadruple (R1, A1, R2, A2) corresponds to a joining pair R1.A1 = R2.A2, where the ref-link is pointing from R1 to R2. If R1.A1 maps to O1.AN1, R2.A2 maps to O2.AN2, the *inclusive class* of R1 maps to the subclass O3 of O1, and the *inclusive class* of R2 maps to the subclass O4 of O2, then the result of the method *path* will be the quadruple (O3,AN1,O2,O4). AN1 is an attribute of O3, which could be explicit or inherited attribute. O2 is the object referenced by the attribute AN1. O4 is a specific subclass of O2 and the XSQL query returns this subclass. The corresponding XSQL query is O3.AN1[O4]. Note that this rule uses the previous rule in group Ident-Class-Hier to get the value for the method sub-class of TQobj.
  Consider the joining pair Register.Ssn = GradStudent.Ssn. Observe that the relations Register and the relation GradStudent correspond to the classes Register-class and Grad-class respectively, so the instantiation of the mapping rules will be as follows:

TQobj(CRrel,d_2)[path@(Register,Ssn,GradStudent,Ssn) $\rightarrow$
(Register-class,Sid,Student-class,CsGrad-class)] $\leftarrow$
CRrel[reference $\leftrightarrow$ (Register,Ssn,GradStudent,Ssn)] $\wedge$
HTrel-objMapping(Register,d_2)[mappings $\leftrightarrow$ map(Register,Ssn,d_2)
[object-attr-domain $\leftrightarrow$ (Register-class,Sid,Student-class)]]$\wedge$
HTrel-objMapping(GradStudent,d_2)[
mappings $\leftrightarrow$ map(GradStudent,Ssn,d_2)
[object-attr-domain $\leftrightarrow$ (Grad-class,Sid,integer)]] $\wedge$
HTrel-objMapping(GradStudent,D) [
mapped-class $\leftrightarrow$ Grad-class[super-class $\leftrightarrow$ Student-class]] $\wedge$
TQobj(CRrel,d_2)[sub-class@Register $\rightarrow$ Register-class] $\wedge$
TQobj(CRrel,d_2)[sub-class@GradStudent $\rightarrow$ CsGrad-class]

References

Ahmed, R., J. Albert, W. Du, W. Kent: An overview of Pegasus. Proceedings of the International Conference on Data Engineering. (1993)

Albert, J., R. Ahmed, M. Ketabchi, W. Kent, M. Shan: Automatic importation of relational schemas in Pegasus. Proceedings of the International Conference on Data Engineering. (1993)

Arens, Y. and C. Knoblock: Planning and reformulating queries for semantically-modeled multidatabase systems. Proceedings of the International Conference on Knowledge Management. (1992)

Dorr, Bonnie J.: Machine Translation: A View from the Lexicon. MIT Press. Cambridge, MA. (1993)

Kifer, M. and G. Lausen: F-logic: A higher-order language for reasoning about objects, inheritance and scheme. Proceedings of the ACM Sigmod Conference. (1989)

Kifer, M., W. Kim, and Y. Sagiv: Querying object-oriented databases. Proc. of the ACM Sigmod Conference. (1992)

Kim, W., Choi, I., Gala, S. and Scheevel, M.: On resolving heterogeneity in multidatabase systems. Distributed and Parallel Databases, Vol. 1. (1993) 251–279

Lawley, M.: A Prolog interpreter for F-logic. Technical report. Griffith University. (1993)

Lefebvre, A., P. Bernus and R. Topor: Query transformation for accessing heterogeneous databases. Joint International Conference and Symposium on Logic Programming, Workshop on Deductive Databases. (1992)

Qian, X.: Semantic interoperation via intelligent mediation. Workshop on Research Issues in Data Engineering. (1993)

Raschid, L., Chang, Y. and B. Dorr: Interoperable Query Processing with Multiple Heterogeneous Knowledge Servers. Proceedings of the Second International Conference on Information and Knowledge Management. (1993)

Raschid, L., Chang, Y. and B. Dorr: Query Transformation Techniques for Interoperable Query Processing in Cooperative Information Systems. Proceedings of the Second International Conference on Cooperative Information Systems. (1994)