1 Introduction and comparison with related research

One of the most important requirements of a cooperative information system (CoopIS) is to support interoperable query processing, when multiple heterogeneous servers are being 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 in the CoopIS. After such a transformation, a query may be answered using information from multiple sources. In this paper, 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 global dictionaries, and applying a set of mapping rules to obtain a transformation.

This research can be placed in the context of other research in the area of *representational heterogeneity*, where equivalent information is stored in multiple schema using different models, leading to possible mismatch. However, since the knowledge is equivalent, it is assumed that it is always possible to resolve any mismatch in the schema. Many approaches advocate support of a global schema (see a summary in Batini *et al* (1986). However, past research in schema conversion was often ad hoc and mapping information to resolve discrepancies was not represented declaratively. Global schema and query transformations were handcrafted. A disadvantage of the global schema approach is that a unified schema needed to be accepted by all users; this may not be feasible in a CoopIS environment. Further, most research only address the issue of accessing local schema via queries posed against the global schema. Little work has been done on interoperable query processing with local queries (existing legacy applications) that may wish to access information in other servers, as envisioned in a CoopIS.

Kent (1991) classifies mismatches in heterogeneous databases as domain mismatches (discrepancies among data types, values, units of measure, etc.) and schema mismatch (different constructs in the data models representing the same concept). They use integrator functions to solve the mismatch. They do not address the problem of query transformation. Discrepancies that may occur within the same data model is investigated by Krishnamurthy *et al* . (1991). Kim and Seo (1991) provide an exhaustive list of conflicts that may occur in a multidatabase environment, expressed in the context of the relational data model. Ahmed *et al* (1993), Albert *et al* (1993) describe Pegasus, a heterogeneous DBMS that provides access to native and external heterogeneous (relational) schema. Here, too, queries can only access the local schema via the *imported* Pegasus global schema. Finally, in SIMS (see Arens and Knoblock (1992)), information sharing from multiple relational schema is facilitated through using the LOOM knowledge representation schema to construct a global schema for each application domain. Queries are proposed against the global schema and the SIMS system formulates a plan for query evaluation. This research, too, does not address the issues of interoperable query processing for local queries.

In contrast, our approach does not require schema integration among the multiple servers in a CoopIS. We also address the problem of transforming queries transparently, so the local user (or existing legacy application) does not need explicit knowledge (of a global schema or mapping knowledge among the schema) to pose a query. However, if a unified schema was available in the CoopIS environment, then we can provide interoperability wrt the unified schema.

Research in combining deductive and object-oriented data models is also relevant (see, *e.g.*, Barsalou and Gangopadhay (1992), Gardarin and Valduriez (1992), and Jeusfeld and Jarke (1991)). The Conceptbase system by Jeusfeld and Jarke (1991) expresses object-oriented constructs in the form of a deductive database system. Gardarin and Valduriez (1992) propose a language that is suitable for relational, deductive and object-oriented data models. Although the goals of these researchers are different, they provide insight into knowledge that is relevant to the query transformation process.

The research that is closest to our work is described by Lefebvre *et al* (1992) and Qian (1993). Lefebvre *et al* (1992) consider the the problem of interoperable query processing, but their approach is limited to relational schema. Our work can be seen as an extension of their approach, to include heterogeneous models. Qian (1993) suggests that a language which has minimal *representation bias* may express mismatch in representation among heterogeneous schema. The paper describes an approach to interoperable query processing for different schema based on the same data model (entity relationship model), but should be applicable to heterogeneous data models as well.

Our approach will encapsulate the information on resolving representational heterogeneity in a knowledge dictionary and use this to support query mapping and query transformation. Users will not need to write their queries against a global schema so we provide interoperability in a transparent manner. In addition, the knowledge dictionary represents the mapping and transformation information in a declarative manner and can be also treated as a knowledge source, to be used to provide explanations, etc. We use a high level logic language, F-logic (see Kifer and Lausen (1989), Kifer *et al* (1990), Lawley (1993)). Finally, our approach also extracts relevant semantics from a source query and the transformed queries may reflect the user's requirements more closely and may be more efficient to process.

In this paper, we use the example of transforming a source XSQL query, posed against an object schema, to a target SQL query for an equivalent relational schema, to describe the various tasks involved. This paper is organized as follows: section 2 gives examples of transforming from XSQL queries to SQL queries. Section 3 provides an architecture for interoperable query processing. Section 4 describes the function of the Extractor Module (EM) operating on a source XSQL query and section 5 discussed a heterogeneous mapping from an object schema to an equivalent relational schema. Section 6 briefly outlines the F-logic implementation. In a companion paper, we have described the transformation from a relational SQL query to an XSQL object query (see Raschid and Chang (1994)).

2 Transforming Queries from Object to Relational Schema

We use a sample object schema (appendix **??**) and *equivalent* relational schema (appendix **??**). In the object schema,[1] each node is an object, and the broken arcs represent an inheritance hierarchy, A solid arc represents a pointer to another object, for example, LeasedVehicle is an attribute of Company and refers to some object Vehicle.

The source query for the object schema is an XSQL-like query.[2] We use simple XSQL queries that only differ from SQL queries in the query path expressions (qpe) of the where clause. An XSQL query takes the following form:

Query 1

Select Z
from Company X, GasolineAuto Y, Company Z $\cdots$
where X.LeasedVehicle[Y].Manufacturer[Z] $\cdots$

In the object schema, we interpret X.LeasedVehicle [Y] as selecting each Company X having LeasedVehicle Y, where Y is restricted to be a GasolineAuto (in the from clause). The Company Z is the Manufacturer of the GasolineAuto. GasolineAuto does not have an attribute Manufacturer and has to inherit this from the class Vehicle.

There are two relational schemas in appendix **??**. In RDBMS #1, relations RelCompany-Vehicle and RelYachtClub-Vehicle correspond to vehicle information for Company and YachtClub,

[1] The objects described in the schema are *class* objects which may be distinguished from *individual* objects.

[2] XSQL is an extension of SQL to access object data bases and we refer the reader to Kifer *et al* (1992) for details of the language.

RelVehinfo has the Model and Manufacturer information on each Vehicle and RelAutoInfo and RelBoatInfo correspond to Auto and Boat, respectively. To obtain a query equivalent to *Query 1* in RDBMS #1, we use the knowledge that the VehicleId values in RelCompany-Vehicle may refer to either boats or automobiles. We use these VehicleId values to refer to information in RelAutoInfo. We must select only those tuples in RelAutoInfo corresponding to GasolineAuto, as indicated by the value of the Type attribute. To obtain the manufacturer information we need to use the VehicleId values to refer to the relation RelVehInfo. An equivalent SQL query would be as follows:

Query 2

Select Manufacturer **from** RelCompany-Vehicle RelVehInfo RelAutoInfo
where RelCompany-Vehicle.LeasedVehicle = RelAutoInfo.VehicleId and
RelAutoInfo.Type = GasolineAuto and RelAutoInfo.VehicleId = RelVehInfo.VehicleId

Next, we consider the relational schema of RDBMS #2, where information on automobiles (GasolineAuto and DieselAuto) are actually stored in two relations, RelHatchBack and RelSedan. There is also a relation storing information on diesel vehicles, RelDslVeh and a relation RelAllVeh that stores the Manufacturer and Model information for all vehicles. The corresponding SQL query is as follows:

Query 3

Select Manufacturer **from** RelCompany-Vehicle RelHatchBack RelSedan RelDslVeh RelAllVeh
where RelCompany-Vehicle.LeasedVehicle = RelAllVeh.VehicleId
& RelDslVeh.DslVehId = RelAllVeh.VehicleId
& RelSedan.AutoId = RelDslVeh.DslAutoId & RelHatchBack.AutoId = RelDslVeh.DslAutoId
& RelSedan.VehType = DieselAuto RelHatchback.AttrVehType = DieselAuto

3 Architecture for Query Transformation

An architecture for a system that supports interoperable query processing through transparent query transformation is described (see appendix **??** for a figure). It is assumed that local and global knowledge dictionaries that support the mapping among the heterogeneous schema in a CoopIS are represented in a declarative language, in some canonical form. Details of the language, F-logic, and the canonical representation are discussed in a later section. We describe some functional modules of the **Interoperablity Module (IM)**, in detail, and refer to various knowledge dictionaries, as needed.

Corresponding to each DBMS in the CoopIS, there is a local knowledge dictionary that encodes relevant knowledge on (1) the query language constructs, and (2) the data model and the local schema. This local dictionary is accessed by the **Extraction Module (EM)** of the IM. This module accesses both the local query language and schema dictionary, for the task of extracting relevant semantic information from the source query. The EM will represent the semantic information, extracted from the query constructs, in a canonical representation, as a corresponding structure, QueryPathSemantics (QPSobj or QPSrel, resp.). Since the source query could be complex, the semantics may be represented in several such structures.[3]

The next module performs the transformation among the heterogeneous schema and is known as the **HeTerogeneous Mapping (HTM)** . It has several functions, each of which accesses the global knowledge dictionary. The first function is identifying the mapping knowledge, HTMapping, representing the correspondence among entities in the different schema. In the case of a mapping from an object to a relational schema, this knowledge is classified in a structure HTobjrelMapping. For a mapping from a relational to an object schema, this knowledge is classified as HTrel-objMapping.

[3]We note that here we are only considering a simple subset of XSQL queries.

The second function is identifying and applying the correct mapping rule, which determines the correct transformation. To determine this, the HTM accesses information in the corresponding QPS structure and the relevant HTMapping information in the global dictionary. After applying the rules, HTM produces a TransfQuery (TQrel or TQobj, resp.) to represent the target query in the canonical representation. The mapping rules are classified in several groups, each of which performs a class of transformations relevant to the target model/query language. When mapping from an object to a relational schema, the rules are grouped as follows: Ident-Rel-Attr (identifies corresponding relations and attributes), Sel-Constr (identifies selection criteria) and Join-Path (identifies join information). Selected rules in each group will be applied, and each produces a part of TQrel.

Currently, since we only consider a simple XSQL query, and since HTMapping and the mapping rules resolve fairly simple conflicts when going from an object schema to a relational schema, the resulting TQrel structure we obtain is also simple. As we consider more complex source queries (represented by one or more QPS), and resolve more complex conflicts, we expect to extend the HTMapping structure and the classes of mapping rules, to reflect this added complexity and knowledge. Consequently, we also expect to extend the resulting TQ structure (TQrel or TQobj), so that it corresponds to some fairly expressive query in the target schema/query language. We note that if a unified schema were available, then HTM could actually provide interoperability wrt this unified schema.

The final function of the HTM is to combine the various TQobj or TQrel fragments, each of which is obtained from a fragment of the source query. Again, there are rules CombTQobj or CombTQrel, respectively, which would combine the TQobj or TQrel fragments. These combining rules also reflect semantics of the target data model/schema and query language.

The final module of the IM is the **Generator Module (GM)** . This module accesses the local knowledge dictionary, corresponding to the target schema. It may use knowledge on the cost of processing a query, and knowledge of equivalences in the target query language and schema, to produce an optimal query in the target query language.

4 The Extractor Module

We describe the Extractor Module (EM), parsing a source XSQL-like query, to extract information relevant to the query transformation process. A simple XSQL query is as follows:

Query 4
Select
from object varname, object varname $\cdots$
where qpe

The *query path expression* (qpe) in the **where** clause of the query is of the following form:
X.attr-X [Y].attr-Y [Z].attr-Z [A].attr-A [B] $\cdots$
where X, Y, Z, etc., are object classes, attr-X, attr-Y, etc., are attribute labels, and X.attr-X [Y] is read as *the attribute attr-X for the object X, is an object Y*.

The EM, using information in the local knowledge dictionary, will express the relevant semantics extracted from the source query as QPS-obj (or QPS-rel for a source SQL query). For simplicity, we consider an XSQL query with a single (simple) qpe in the **where** clause. Each (simple) QPS-obj structure is as follows:[4]

[**begin-object**, < **sub-list** | **sup-list** >, **end-object**, **attribute**]

[4]When a source XSQL query has a more complex structure, e.g., multiple qpe and nesting, etc., then it will first be converted to some appropriate normal form and will generate multiple QPS-obj structures in the canonical representation. The same is true for a complex SQL source query.

Consider the QPS-obj for [Y].attr-Y in the qpe X.attr-X [Y].attr-Y [Z]. The **begin-object** is $\mathbf{Y}^*$, corresponding to X.attr-X, i.e., the object pointed to by the pair (X, attr-X). If Y^* is identical to object **Y** specified in the **from** clause, then the **sub-list** will be empty. If Y^* differs from Y then **sub-list** will contain a path from Y^* to Y. The **end-object** will be some $\mathbf{Y'}$ which has **attribute attr-Y**. If Y is the same as Y', then, the **sup-list** will be empty; otherwise, sup-list will contain the path from Y to Y'. We refer the reader to Raschid *et al* (1993) for details. We use the following table to summarize a procedure **Extract** which is performed by the EM and describe it using examples:

Extract : Extracting information from X.attr-X[Y].attr-Y[Z]		
$Extract_1$	Object type of Y known in **from**	
	$Extract_{1,1}$	X.attr-X points to object Y^* and $Y^* \equiv Y$
	$Extract_{1,2}$	X.attr-X points to object Y^* and $Y^* \neq Y$
$Extract_2$	object type of Y not known in **from**	
	$Extract_{2,1}$	X.attr-X points to object Y^* and Y^* has sub-class(es) Y which have attribute attr-Y
	$Extract_{2,2}$	X.attr-X points to object Y^* but no sub-class(es) Y of Y^* has attrib. attr-Y. However, Y and Y^* may inherit attr-Y from Y'

$\mathbf{Extract_1}$ – object type of Y known in from

$\mathbf{Extract_{1,1}}$ – if $\mathbf{Y \equiv Y^*}$

Consider the following example *Query 5:*

Select * **from** GasolineBoat Y
where YachtClub.OwnedVehicle[Y]

The object pointed to by (YachtClub,OwnedVehicle), or Y^*, is exactly the object GasolineBoat, specified for Y in the query. Thus, in the corresponding QPS-obj, GasolineBoat is the begin-object and the sub-list will be empty.

$\mathbf{Extract_{1,2}}$ – if $\mathbf{Y \neq Y^*}$ but is known in from

Consider the following example *Query 6:*

Select * **from** DieselBoat Y Company Z
where Company.LeasedVehicle [Y].Manufacturer [Z] $\cdots$

The object pointed to by (Company, LeasedVehicle), or Y^* is Vehicle, but Y is specified to be DieselBoat. Now, in the QPS-obj, the begin-object is Vehicle and sub-list will contain the path from Vehicle to GasolineBoat. The end-object (that has attribute Manufacturer) is Vehicle and GasolineBoat inherits this attribute from Vehicle. Normally, the sup-list will contain the path from GasolineBoat to Vehicle. However, here, the sup-list is empty, since sub-list already traverses this path (see Raschid *et al* (1993) for details).

$\mathbf{Extract_2}$ – object type of Y unknown in from

$\mathbf{Extract_{2,1}}$

Consider the following example *Query 7:*

Select * **from**
where Company.LeasedVehicle [Y].Registration $\cdots$

Here, Y is unspecified in the **from** clause. The object Y^* which is pointed to by (Company,LeasedVehicle) is Vehicle. From the object schema, **Extract** will infer that there are two objects (Y), namely DieselBoat and GasolineBoat, with attribute Registration. Thus, in QPS-obj, there are two paths in sub-list (from Vehicle to DieselBoat, and from Vehicle to GasolineBoat).

$\mathbf{Extract_{2,2}}$

Consider the following example *Query 8:*

Select * **from**
whereYachtClub.OwnedVehicle [Y].Manufacturer [Z].attr-Z

Here, Y is unspecified in the from clause. The object Y^* which is pointed to by (YachtClub,OwnedVehicle) is GasolineBoat. However, GasolineBoat does not have attribute Manufacturer

nor does it have any sub-class with this attribute. However, GasolineBoat can inherit the attribute Manufacturer from object Vehicle, or Y′. Thus, for the QPS-obj, begin-object is GasolineBoat, the sub-list is empty, the end-object is Vehicle, and the sup-list is a path from GasolineBoat to Vehicle.

5 Mapping Information from Object to Relational Schema

We now briefly discuss the function of the HTM - the module that accesses the HTMapping global dictionary and applies the appropriate mapping rule to obtain a transformation among the heterogeneous schema. We focus on the HTobj-relMapping here. The canonical representation of HTobj-relMapping is expressed as a mapping from a particular object, say a *start object* (SO) and corresponding *attribute*. If the attribute is not actually structurally associated with the start object, then it must be associated with an *end object* (EO) through a class hierarchy. Finally, we may also refer to *path objects* (PO) occurring in the path between SO and EO. This combination (SO, EO, PO, attribute) maps to some attribute in the relational schema.[5] If we consider a simple QPSobj structure, there may be either one or two such combinations, depending on whether the sub-list and sup-list are empty lists. To simplify the explanation, we only consider HTobj-relMapping knowledge relevant to a single combination of (SO, EO, PO, attribute) and refer the reader to (Raschid *et al* (1993) for details on the combining rules CombTQobj.

Figure 1 summarizes the different cases that correspond to the HTM function for the obj-rel transformation. We discuss a few of the cases using examples.

Case 1	Transformation depends on start object S0	
	Case 1.1	(S0, attrib.) maps to list {(rel. name RN, attrib. name AN)}
	Case 1.2	(SO, attrib.) maps to {(rel. name RN, attrib. name AN, crit. to select subset of RN tuples based on S0)}
Case 2	Transformation depends on end object E0	
	Case 2.1 Does not depend on S0	(E0, attrib.) maps to list { (rel. name RN, attrib. name AN)}
	Case 2.2 Also depends on S0	(EO, attrib., SO) maps to list (rel. RN, attrib. AN, crit. to select subset of RN tuples based on S0)}
	Case 2.3 Does not depend on SO	(EO, attrib.) maps to list {RN, AN, crit. to select subset of RN tuples based on EO)}
	Case 2.4 Also depends on S0	(EO, attrib., SO) maps to {(RN, AN, crit. to select RN tuples based on EO, crit. to select RN tuples based on E0)}
Case 3	Transformation depends on path objects PO in path from S0 to E0 and EO	
Case 4	General case where (EO, attrib.) points to an object - requires combining rules	

Figure 1: Cases Describing the Functionality of HTM From Object to Relational Schema

Case 1.1 – (SO, attrib.) maps to {(RN, AN)}
Consider an example pair from the object schema of appendix **??**, say (DieselAuto, Model). This will produce an example mapping such as {(RN ≡ RelDieselAuto, AN ≡ AttrModel)}.
Case 1.2 – (SO, attrib.) maps to {(RN, AN, subset of tuples of RN based on SO)}
The same pair (DieselAuto, Model) may map to {(RN ≡ RelAuto, AN ≡ AttrModel, AttrType = {"DieselHatchBack", "DieselSedan"})}. Here, certain tuples of relation RelAuto are selected, based on the value of AttrType matching the set {"DieselHatchBack", "DieselSedan"}.
Case 2.1 – (EO, attrib.) maps to {(RN, AN)}
Consider an example where SO is DieselVehicle, EO is Vehicle and attrib. is Manufacturer. The

[5] When SO and EO correspond to the sub-list of the QPSobj structure, then the attribute is optional.

mapping produces {(RelVehicle, AttrManufacturer)}. This implies that either all tuples in relation RelVehicle correspond to the DieselVehicle, or there is some other transformation that may be applied to select only those tuples of RelVehicle corresponding to DieselVehicle.
Case 2.2 – (EO, attrib., SO) maps to {(RN, AN, subset of tuples of RN based on SO)}
Consider (DieselVehicle, Mnufacturer, Vehicle) as before, producing {(RelVehicle, AttrManufacturer, AttrType = {"DieselAuto", "DieselBoat"})}. Here, all Vehicles are in RelVehicle, and corresponding to the SO DieselVehicle, we select a subset of RelVehicle tuples, where the value of AttrType in {"DieselAuto", "DieselBoat"}, identifies DieselVehicle.
Case 2.4 – (EO, attrib., SO) maps to {(RN, AN, subset of RN tuples based on SO, subset of RN tuples based on EO)}
Consider (DieselVehicle, Manufacturer, Vehicle) producing {(RelAllItems, AttrManufacturer, AttrEngineType = {"enumeration of Vehicle engine types"}, AttrFuelType = {"enumeration of fuel types for DieselVehicle"})}. Here, all manufacturer information is in RelAllItems. The attribute AttrEngineType selects tuples corresponding to Vehicle and AttrFuelType identifies DieselVehicle

6 Representation of the Heterogeneous Mapping Information

We use a high-level logic to represent the HTMapping knowledge and the mapping rules. We chose the F-logic language as formulated by Kifer and Lausen (1989) and Kifer *et al* (1990). F-logic has good modeling constructs for representing mapping knowledge, it can represent object schema fairly concisely, and there is an interpreter for the language (see, *e.g.*, Lefebvre *et al* (1993)). We briefly present the syntax and semantics of F-logic and then describe the data structures and a few of the transformation rules. Raschid *et al* (1993) provide additional details.

6.1 Syntax of F-logic

We borrow this brief description from Lefebvre *et al* (1992). 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 $\leftrightarrow$ indicates a set-valued method.[6] Note that other values could also be members of this set, and that the expression above does *not* restrict the value of m' for o to be $\{v', v''\}$; it only indicates *some of* the values of the corresponding set. An object can be denoted by a constant, or a term. For example, $dept(cs)$ is a valid object identifier. An *atomic data term* is a data term referring to only one method. 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 $o[m \rightarrow v] \wedge o[n \rightarrow v']$; the expression $o : c[m \rightarrow v]$ is equivalent to $o : c \wedge o[m \rightarrow v]$.

An 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. Deduction rules can be used in a way similar to Datalog rules(or Prolog clauses), i.e., the head of a rule defines a derived method, the value of which can be found by evaluating the body.

6.2 Data Structures for Heterogeneous Mapping

We present the following data structures QPSobj, HTobj-relMapping and TQrel, corresponding to the transformation from an object schema to a relational schema:

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

QPSobj(i)[**begin-object** → X, **sublist** → P_{sub}, **suplist** → P_{sup}, **attr-X** → A]

HTrel-objMapping(X, D)
[**reachable-attr** ↔ A, **includes** ↔ X′,
mappings ↔ map(X, A, D)[rel-attr ↔ (R, Z)],
constraints ↔ constraint(R, X) [attr → AN-C, range → S]
join ↔ joinmap(X, X′, D)[rel-attr ↔ (R_1, R_2, Z_1, Z_2)]]

TQrel(QPSobj, D) [**rel-attr** ↔ (R, AN), **constraints@R,X** ↔ (R, AN-C,S), **join-path@P** → L]

Query path semantics (QPS) represents the semantics extracted from a simple XSQL query path expression. The structure was discussed in detail in section 4.[7] HTrel-objMapping captures the het-map information between the object-oriented and relational schema. Its detailed structure will be explained when we discuss the mapping rules, in the next section.

TQrel corresponds to the canonical representation of the transformed query. It will be used to generate the appropriate relational query for a particular database schema D (the actual generation is not discussed here). The method *rel-attr* will return one or more pairs of relation and attribute names. For each pair that is returned, if there is additional selection criteria, then the method *constraints* returns the appropriate attribute name and the range of values for the selected tuples to qualify. The *join-path* is a method that provides the join information for Case 3 of the HTM obj-rel function. Other structures, e.g., lists are omitted. The rules CombTQrel, for combining the TQrel structures, when an attribute of an object is a pointer to another object, must guide the generation of the relational query as well.

6.3 Heterogeneous Mapping Algorithm and F-logic Rules

The mapping rules that we present here provide a flavor of the types of transformations that can be expressed and are not intended to represent a complete algorithm for transforming XSQL queries. We present rules for two examples of the HTM obj-rel mapping, Case 1.2 and Case 2.4. For the HTM(obj-rel) function, **Ident-Rel-Attr** is a group of rules identifying the corresponding relation and attribute name in TQrel, **Sel-Constr** is a group of rules which specifies selection criteria that must first be satisfied by the tuples of the relations specified in Ident-Rel-Attr rules (or the Join-Path rules). Finally **Join-Path** is a group of rules which specify relations which must be joined, corresponding to the more complicated Case 3 of the HTM(obj-rel) transformation. We do not have any examples of JoinPath rules here, nor of the CombTQrel rules which combine multiple TQrel (see Raschid *et al* (1993)).

Case 1.2

As an example for Case 1.2, (DieselAuto, Model) must provide the following:
{(RelAuto, AttrModel, AttrType = {"DieselHatchBack", "DieselSedan"})}.

The model information on DieselAuto is stored in an attribute AttrModel of a relation RelAuto. The DieselAuto are distinguished by a value of "DieselSedan" or DieselHatchBack" in the attribute AttrType, and in order to select only those tuples we use one of the Sel-Constr rules. A particular instantiation for a rule from Ident-Rel-Attr (which produces the pair (RelAuto, AttrModel) and a rule from Sel-Constr (which selects tuples from RelAuto based on AttrType values) are as follows:

TQrel(i, QPSobj(i), D)[rel-attr ↔ (**RelAuto**, **AttrModel**)] ←
QPS[begin-object → DieselAuto , attr → Model sublist → NULL] ∧
HTobj-relMapping(DieselAuto, D)

[7] The actual structure is more complex since we must ensure that paths in the class hierarchy are not traversed redundantly.

[reachable-attr ↔ Model,
mappings ↔ map(DieselAuto, Model, D) [rel-attr ↔ (RelAuto, AttrModel)]] ∧
TQrel(i,QPSobj(i),D)[constraints@RelAuto, DieselAuto ↔ C] ∧
TQrel(i,QPSobj(i),D)[joining-path@NULL → NULL]

TQrel(i, QPSobj(i), D)[constraints@RelAuto, DieselAuto ↔
(**RelAuto**, **AttrType**, {"**DieselHatchBack**", "**DieselSedan**"})] ←
HTobj-relMapping(DieselAuto, D)
[constraints ↔ constraint(RelAuto, DieselAuto)
[attr → AttrType, range → { "DieselHatchBack", "DieselSedan"}]]

After these two rules are applied, we produce the following TQrel:

TQrel(i, QPSobj(i), D)
[rel-attr ↔ (**RelAuto**, **AttrModel**)
constraints@RelAuto,DieselAuto ↔ **RelAuto**, **AttrType**, {"**DieselHatchBack**", "**DieselSedan**"})]

It should be straightforward to generate an SQL query which selects AttrModel values for those tuples of RelAuto, such that the range of values of AttrType is {"DieselHatchBack", "DieselSedan"}.

Case 2.4

As an example for Case 2.4, (DieselVehicle, Manufacturer, Vehicle) must provide the following: {(RelAllItems, AttrManufacturer, AttrEngineType = {"enumeration of vehicle engine types"}, AttrFuelType = {"enumeration of fuel types for DieselVehicle})}.

The relation RelAllItems stores information on the object DieselVehicle as well as on other vehicles. Based on the attribute AttrEngineType, we select Vehicle tuples, and based on the attribute AttrFuelType, we select DieselVehicle tuples, from RelAllItems. The necessity for having two selection constraints may be necessitated by the fact that DieselVehicle may participate in multiple hierarchies, which may not include Vehicle, and vice versa. The instantiated rules from Ident-Rel-Attr and Sel-Constr are as follows:

TQrel(i,QPSobj(i),D)[rel-attr ↔ (**RelAllItems**,**AttrManufacturer**)] ←
QPSobj(i)[begin-object → DieselVehicle, suplist ↔ P_{sup},
end-object@P_{sup} → Vehicle , attr → Manufacturer, sublist → NULL] ∧
not(HTobj-relMapping(DieselVehicle,D)[reachable-attr ↔ Manufacturer]) ∧
HTobj-relMapping(Vehicle,D)[reachable-attr ↔ Manufacturer,
mappings ↔ map(Vehicle,Model,D)
[rel-attr → (RelAllItems, AttrManufacturer)]] ∧
HTobj-relMapping(DieselVehicle,D)[constraints ↔ constraint(RelAllItems,DieselVehicle)
[attr → AttrEngineType, range → S]] ∧
TQrel(i,QPSobj(i),D)[constraints@RelAllItems,DieselVehicle ↔ C] ∧
TQrel(i,qPSobj(i),D)[constraints@RelAllItems,Vehicle ↔ C′] ∧
TQrel(i,QPSobj(i),D)[joining-path@NULL → NULL]

The following two rules are evaluated in the group Sel-Constr to provide the constraints on RelAllItems for DieselVehicle and Vehicle:

TQrel(i,QPSobj(i),D)[constraints@RelAllItems,DieselVehicle ↔
(**RelAllItems**, **AttrEngineType**,{"**enumeration of vehicle engine types**"})] ←
HTobj-relMapping(DieselVehicle,D)[constraints ↔ constraint(RelAllItems,DieselVehicle)
[attr → AttrEngineType,
range → {"enumeration of vehicle engine types"}]]

TQrel(i,QPSobj(i),D)[constraints@RelAllItems,Vehicle ↔

(**RelAllItems**, **AttrFuelType**, {"**enumeration of diesel vehicle fuel types**"})] ←
HTobj-relMapping(Vehicle,D)[constraints ↔ constraint(RelAllItems,Vehicle)
[attr → AttrFuelType,
range → {"enumeration of diesel vehicle fuel types"}]]

The following TQrel is produced from which an SQL query is easily obtained:

TQrel(i,QPSobj(i),D)[rel-attr ↔ (**RelAllItems**,**AttrManufacturer**)
constraints@RelAllItems,DieselVehicle ↔
(**RelAllItems**, **AttrEngineType**,{"**enumeration of vehicle engine types** "})
[constraints@RelAllItems,Vehicle ↔
(**RelAllItems**, **AttrFuelType**, {"**enumeration of diesel vehicle fuel t ypes**"})]

7 Summary

We have proposed a technique for interoperable query processing in a CoopIS, where a user can pose a query against a local schema, without any knowledge of the other models/schema, such as mappings among schema and conflicts. We used the task of extracting knowledge from an XSQL-like query, and mapping from an object to a relational schema, to produce a transformed relational query, to illustrate the different functions performed by the IM. We use a high-level declarative language, F-logic, to represent the local and global knowledge dictionaries and the mapping knowledge (rules). In a companion paper (Raschid and Chang (1994)), we discuss the transformation from a source SQL query to a target XSQL query against an object schema.

8 Bibliography

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

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

Batini, C., Lenzerini, M. and Navathe, S.B. (December 1986) "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys, Vol. 18, No. 4*, 323-364.

Gardarin, G. and P. Valduriez (1992) "ESQL2: An object-oriented SQL with F-logic semantics," *Proceedings of the International Conference on Data Engineering.*

Jeusfeld, M. and M. Jarke (1991) "From relational to object-oriented integrity simplification," *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases.*

Kent, W. (1991) "Solving domain mismatch and schema mismatch problems with an object-oriented database programming language," *Proceedings of the International Conference on Very Large Data Bases.*

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

Kifer, M., G. Lausen, and J. Wu (1990) "Logical foundations of object-oriented and frame-based languages," , Technical report 90/14, SUNY at Stonybrook.

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

Kim, W. and J. Seo (December, 1991) "Classifying schematic and data heterogeneity in multidatabase systems," *IEEE Computer*, pages 12–18.

Krishnamurthy, R., W. Litwin, and W. Kent (1991) "Language features for interoperability of databases with schematic discrepancies," *Proceedings of the ACM Sigmod Conference.*

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

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

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

Raschid, L., Chang, Y. and B. Dorr (1993) "Query transformation among heterogeneous data models: a prototype based on F-logic," *In preparation.*

Raschid, L. and Chang, Y. (1994) "Transforming queries from a relational schema to an equivalent object schema: a prototype based on F-logic," *Under review.*